CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Optimization of Semantic Cache Query Processing System

by

Munir Ahmad

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Computing Science

Department of Computer Science

August 2017

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD
Expressway, Kahuta Road, Zone-V, Islamabad
Phone:+92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: https://www.cust.edu.pk

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled "**Optimization of Semantic Cache Query Processing System**" was conducted under the supervision of **Dr. Muhammad Abdul Qadir**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science.** The open defence of the thesis was conducted on **06 July, 2017.**

Student Name :		Mr. Munir Ahmad (PC091006)	[illegible]

The Examination Committee unanimously agrees to award PhD degree to Mr. Munir Ahmad in the field of Computer Science.

Examination Committee :

(a)	External Examiner 1:	Dr. Ejaz Ahmed, Associate Professor FAST -NU Islamabad	[illegible]
(b)	External Examiner 2:	Dr. Khalid Saleem, Assistant Professor QAU, Islamabad	[illegible]
(c)	Internal Examiner :	Dr. Muhammad Tanvir Afzal Associate Professor CUST, Islamabad	[illegible]
Supervisor Name :		Dr. Muhammad Abdul Qadir Professor CUST, Islamabad	[illegible]
Name of HoD :		Dr. Nayyer Masood Professor CUST, Islamabad	[illegible]
Name of Dean :		Dr. Muhammad Abdul Qadir Professor CUST, Islamabad	[illegible]

AUTHOR'S DECLARATION

I, **Mr. Munir Ahmad (Registration No. PC091006)**, hereby state that my PhD thesis titled, '**Optimization of Semantic Cache Query Processing System**' is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.

[illegible]

(Mr. Munir Ahmad)

Dated: July, 2017

Registration No : PC091006

PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled "**Optimization of Semantic Cache Query Processing System**" is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

[illegible]

(Mr. Munir Ahmad)
Registration No. PC091006

Dated: July, 2017

Acknowledgements

I pay my sincere thanks to my supervisor who supported me in my thesis with his patience and knowledge. I attribute the completion of my degree to his encouragement and effort and without him this thesis would have not been possible. One simply could not wish for a better or gracious supervisor.

I am profoundly thankful to CDSC research group for their guidance, counselling and tremendous help which contributed to give final form to my work.

I am also thankful to HEC for providing me the opportunity to play a role in contributing to the knowledge base of Pakistan. May our country prosper.

List of Publications

It is certified that following publications have been made out of the research work that has been carried out for this thesis.

Journal Papers

1. Ahmad, M., Qadir, M.A., and Ali, T. (2017). Indexing for Semantic Cache to Reduce Query Matching Complexity. Journal of the National Science Foundation of Sri Lanka, 13-22 March 2017.

Conference Papers

1. Ahmad, M., Asghar, S., Qadir, M. A., and Ali, T. (2010). Graph Based Query Trimming Algorithm for Relational Data Semantic Cache. *The International Conference on Management of Emergent Digital EcoSystem, MEDES10*, 2010. Pages-47-52

2. 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.*

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

4. Ahmad, M., Qadir, M.A., Razzaque, A., and Sanaullah, M. (2008). 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.

Book Chapters

1. Ahmad, M., Qadir, M. A., and Ali, T., Abbas, M. A., Afzal, M.T. (2012). Semantic Cache System. *In Semantics – Book 2, ISBN 980-953-307-286-4, Published by INTEC – Open Access Publisher*, 2012.

August 2017 Signatures

Munir Ahmad
Reg. No PC091006

Abstract

High availability and low latencies of data are major requirements in accessing contemporary large and networked databases. However, it becomes difficult to achieve high availability and reduced data access latency with unreliable connectivity and limited bandwidth. These two requirements become crucial in ubiquitous environment when data is required all the times and everywhere. Disconnectivity with data is more likely with multimedia streaming, especially with frequent queries and low bandwidth. Inter-queries common data is one of the important qualities of consecutive queries. Cache is one of the promising solutions to improve availability and reduce latencies of remotely accessed data by storing and reusing the results of already processed similar queries. Conventional cache lacks in partial reuse of already accessed & locally stored data, while semantic cache overcomes the limitation of conventional cache by reusing the data for partial overlapped queries by storing description of queries with results. There is a need of an efficient cache system to improve the availability, reduce the data access latencies and the network traffic by reusing the already stored results for fully and partially overlapped queries. An efficient cache system demands efficient query processing and cache management.

In this study, a qualitative benchmark with four qualities as Accuracy, Increased Data Availability, Reduced Network Traffic and Reduced Data Access Latency is proposed to evaluate a semantic cache system, especially from query processing point of view. The qualitative benchmark is then converted into six quantitative parameters (Semantics and Indexing Structure IS, Generation of Amending Query GoAQ, Zero Level Rejection ZLR, Predicate Matching, *SELECT_CLAUSE* Handling, Complexity of Query Matching CoQM) that help in measuring the efficiency of a query processing algorithm. As the result of evaluation, it is discovered that existing algorithms for query trimming can be optimized.

Architecture of a semantic cache system is proposed to meet the benchmark criteria. One of the important deficiencies observed in the existing system is the storage of query semantics in segments (indexing of the semantics) and the organization of these segments. Therefore, an appropriate indexing scheme to store the semantics of queries is needed to reduce query matching time. In the existing indexing schemes the number of segments grows faster than exponential, i.e., more than 2^n. The semantic matching of a user query with number of segments more than 2^n will be exponential and not feasible for a large value of n. The proposed schema-based indexing scheme is of polynomial time complexity for the matching process.

Another important deficiency observed is the large complexity of query trimming algorithm which is responsible to filter the semantics of incoming query into local cache and remote query. A rule based algorithm is proposed for query trimming that is faster and less complex than existing satisfiability/implication algorithms. The proposed trimming algorithm is more powerful in finding the hidden/implicit semantics, too.

The significance of the proposed algorithms is justified by examples/case studies in comparison with the previous algorithms. Correctness of proposed algorithms is tested by implementing a prototype and executing a number of case studies. The final outcomes revealed that the proposed scheme has achieved sufficient accuracy, increased availability, reduced network traffic, and reduced data access latency.

Contents

List of Figures

List of Tables

Abbreviations

GoAQ	Generation of Amending Query
sCacheQP	Semantic Cache Query Processing
ZLR	Zero Level Rejection

To my family and my supervisor…

Chapter 1

Introduction

This chapter presents the introduction about the domain, problem statement, research methodology is and some definitions and notations.

1.1 Introduction to the domain

One of the economical ways to develop a very large-scale database is to distribute it among multiple server nodes [1]. The main problem with these types of systems is slow retrieval of data due to added network latency; especially when network or server load is very high [2, 3]. There could be many clients (may be in 1000s accessing a popular website) accessing the server with the same data at the same time. The example below provides a clear overview regarding the understanding of accessing the server with the same data at the same time.

Assume that, four users search the nearest restaurants while staying in Rawalpindi city Pakistan. The results of the searched query are shown in 1.1.

As in 1.1, it is depicted that same data is accessed by multiple users. The challenge is to minimize the access time in order to show the result immediately. There exist many techniques that minimize the access time. One technique is the replication of data on many servers and the other is the use of cache near the client. Energy could also be saved by using a cache [4].

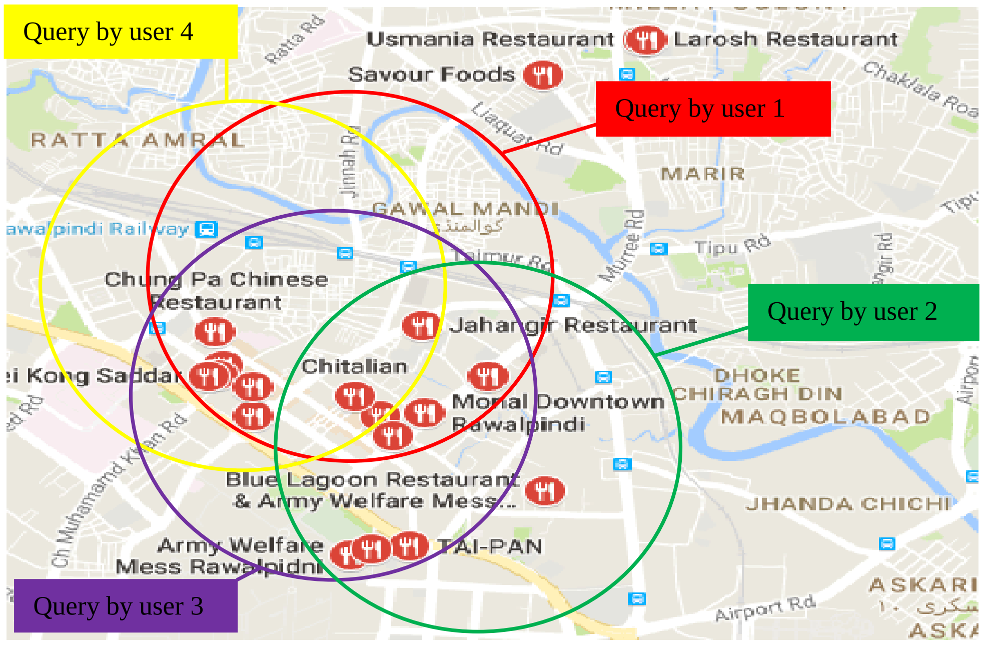

FIGURE 1.1: Nearest Restaurants

Use of a cache near the client is a cost-effective solution especially for the repeated requests of the same data and can be used with or without replication [5]. The cache can be designed at different levels by using different techniques.

Historically, a cache was introduced at the processor level (in hardware) to transfer a block of instructions or data from primary storage to processor to speed-up the execution of a program [5]. This hardware level cache (in processor) is normally used for small amount of data. For larger blocks of data, a cache may be implemented at the primary storage level (in memory) if the original data is stored in secondary storage. A cache may be organized as secondary or primary storage level [6] on the local machine depending upon the required performance and available resources (e.g., size of primary storage) for a remotely stored data (networked storage).

Different techniques that organize a cache, whether it is processor level, primary storage level, or secondary storage level are associative cache [7], and address-based cache linked with description tags [8, 9]. In associative cache, contents are used to address the location of the data and in address-based cache; the address of a storage location is mapped on a tag. That is, the location of a cache can be accessed by using tags. The description tags describe the contents stored in the corresponding location and it could be syntactic or semantic. In syntactic tags, the data is accessed if the tags for requested contents exactly match with the tags of the stored contents. In semantic tags, the access system would be able to check the meanings of the tags for requested contents (semantics) with the meanings of the tags (semantics) of the stored contents, and then the stored data is accessed according to the matched semantics [10, 11]. In the context of databases cache, the syntactic cache may be organized as page or a tuple cache [12, 13]. The organization which processes the meanings of requested and stored tags is termed as semantic cache.

Database cache [14] is used to reduce the data retrieval time. Typically, in database cache a portion of secondary memory is used to store the already fetched data and reuse the stored data in the future for similar queries [15]. In order to understand the concept of cache in databases and to know the difference between syntactic and semantic cache, the following example would be helpful:

Assume the following query is executed on a relation that has all the required columns and the data retrieved against the query is stored in a cache along with the tag, which is the query itself.

SELECT age, name *FROM* student *WHERE* age< 28

Afterwards, when the exactly same query is submitted, the cache system after matching both the tags (stored and query submitted) can work out that the data is available in the cache and it can be returned against the query. This is an example of a syntactic cache.

Assume a new query is submitted for execution as given below:

SELECT age, name *FROM* student *WHERE* age< 25

This query does not exactly match with the tag of the previously stored data in the cache. Therefore, the syntactic cache system will miss the contents from the cache and send a request to the server. However, we can see the meanings of both the query and tag stored and can easily see that all the data is available in the cache which is described by the query semantics. The semantic cache will return the result from the cache. Working of semantic cache is shown in 1.2. The concept of semantic cache was first introduced to answer the queries partially from local site by Dar [10]. One of the important parameters used to evaluate the performance of a cache is hit ratio, that is, the number of queries answered locally from the cache out of the total number of queries posed.

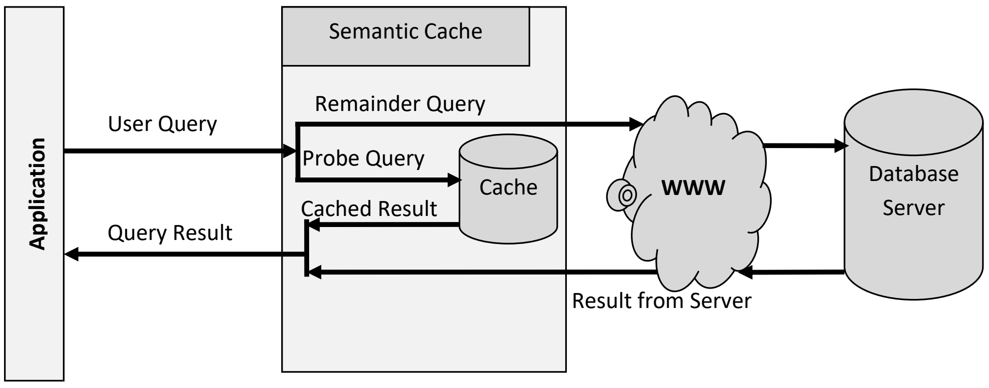

FIGURE 1.2: Working of Semantic Cache

Semantic cache has an ability to increase the hit ratio up to the maximum possible extent [16, 17] as compared to the page and tuple cache [12] due to having the ability to answer partially overlapped queries by using the meanings/semantics. Semantic caching provides significant workload reduction in distributed systems, especially in mobile computing and it improves the overall performance [18]. The claim is still valid as shown by the results of the research presented in this thesis.

There are two major activities to manage a semantic cache; one is query processing and the second is cache management. The efficiency of a semantic cache system depends upon the efficiency of these two activities. Query processing is the process which returns the result against user posed query. Semantic cache query processing is done by dividing a user query into the probe and remainder queries on the basis of query matching with the stored semantics of the cache. In order to understand the working and the significance of a semantic cache system, the following example would be helpful.

Assume the following query is executed on a relation that has all the required columns and the data retrieved against the query is stored in a cache.

$$\begin{gathered} \mathit{SELECT} * \mathit{FROM}\ \text{Employee} \\ \mathit{WHERE}\ \text{age} > 20\ \mathit{AND}\ \text{age} \leq 40\ \mathit{AND}\ \text{salary} > 20000\ \mathit{AND}\ \text{salary} \leq 40000 \end{gathered}$$

Data against above query can be visualized as shown in 1.3.

Assume a new query is submitted for execution as given below:

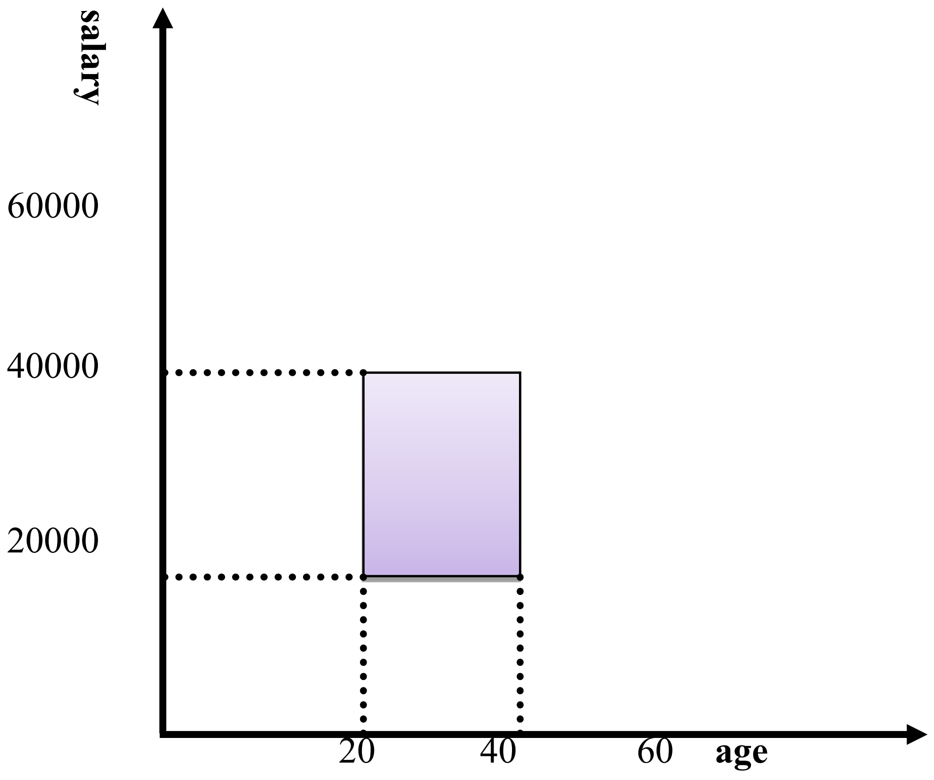

FIGURE 1.3: Data Stored in Cache

$$SELECT\ {*}SELECT\ \text{Employee}$$
$$WHERE\ \text{age} > 35\ AND\ \text{age} \leq 45\ AND\ \text{salary} > 35000\ AND\ \text{salary} \leq 55000$$

Data required in this query is partially overlapped with data stored in semantic cache. Here query will be divided into two sub queries, probe query (data available at cache) and remainder (data retrieved from the server) query as shown in 1.4.

In fact, the efficiency of query processing depends upon the efficiency of the division process (query trimming) of a user query into sub queries (probe and remainder) as well as on retrieval time against both probe and remainder queries. Efficiency of query trimming depends upon the semantic indexing (explained in detail in Chapter 2), too. Actually, semantic indexing is a major activity of cache management. In this context, we can say that efficient semantic caching demands efficient query processing and indexing schemes.

In this research study, our aim is to analyze the query processing and semantic indexing scheme for semantic cache system in terms of efficiency point of view.

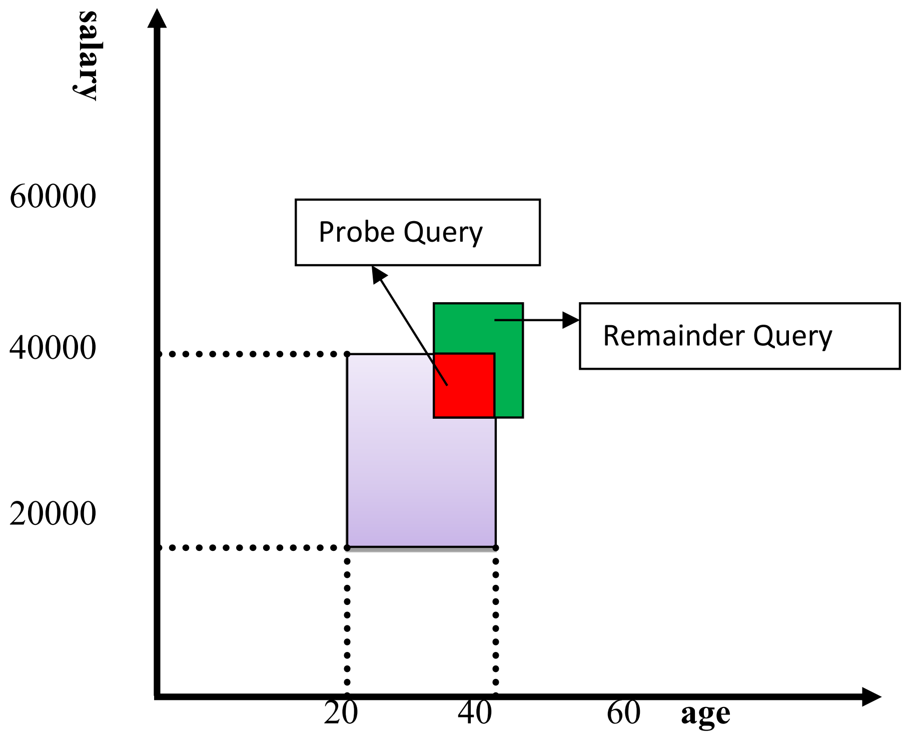

Figure 1.4: Probe and Remainder Query

Moreover, it was also observed that there is a need to define efficiency matrix, which is also performed in this study. Then, the focus would be on the designing of an efficient semantic cache query processing and indexing system.

An efficient semantic caching must ensures following four qualities:

- Accuracy
- Increased Data Availability
- Reduced Network Traffic
- Reduced Data Access Latency

We set these four qualities as a benchmark to evaluate the semantic caching system. These four points of benchmark are qualitative. We have associated some quantitative parameters to measure each of qualitative measure for the semantic cache system. Associated quantitative parameters are discussed in Chapter 2 and all qualitative measures of semantic caching system are discussed below one by one.

Accuracy: A semantic cache system must ensure that there is no extra data included in the result and there is no missing data from the produced result. This quality is termed as the production of the correct result. Correctness of a semantic cache system is achieved by applying the following rule.

$$\text{Data}_{\text{RemainderQuery}} \cup \text{Data}_{\text{ProbeQuery}} \implies \text{Data}_{\text{User\_Query}} \tag{1.1}$$

Rule in Equation 1.1 depicts that the collective data against remainder and probe queries must always be equal to the data being requested by the user query.

Increased Data Availability: As the cache is a local copy of the frequently accessed data, an additional advantage of the increased availability in case of the failure of remote data accesses can be adopted as well. Availability of the system with limited size of the cache can be increased by ensuring that cache does not store redundant data as it is evident by rule given in Equation 1.2.

$$\text{Segment}_{\text{j-cahce}} \cap \text{Segment}_{\text{i-cache}} \implies \varnothing \;\; \forall i, j \text{ when } i \neq j \tag{1.2}$$

Reduced Network Traffic: Data access time would be reduced by the decreased network traffic on average. Less number of remainder queries and the increased hit ratio would result in a less amount of network traffic for the same data access. The following two rules would ensure the decreased network traffic, too. By ensuring following two rules given in Equation 1.3 and 1.4, the network traffic would be minimized too.

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{cached}} \implies \varnothing \tag{1.3}$$

$$\text{Segment}_{\text{i-cache}} \cap \text{Segment}_{\text{j-cache}} \implies \varnothing \; \forall i, j \text{ when } i \neq j \tag{1.4}$$

Reduced Data Access Latency: Data access Latency of query, is the total time used to retrieve the result against the user query. Data access time depends on hit ratio, query trimming algorithm complexity, semantic indexing, network traffic, network bandwidth and size of the database. These parameters are discussed here one by one.

Data access time decreases with the increase of hit ratio as there will be more number of local accesses and less number of remote accesses. Hit ratio would be maximized by ensuring the rules given in Equation 1.5 and 1.6.

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{cached}} \implies \varnothing \tag{1.5}$$

$$\text{Segment}_{\text{i-cache}} \cap \text{Segment}_{\text{j-cache}} \implies \varnothing \; \forall i, j \text{ when } i \neq j \tag{1.6}$$

The first rule says that none of the data against remainder query ever overlapped with cached data and the second rule depicts that data in each segment of cache must be disjoint with the data of the remaining segments of the cache.

Data access time also depends on the time taken by the algorithm that divides the user query into the probe and remainder queries. A faster query trimming algorithm would result into the lesser data access time. The efficiency of a query trimming algorithm depends on its complexity and ensuring rule is given in Equation 1.7 in the implementation of the algorithm.

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{cached}} = \varnothing \tag{1.7}$$

Query trimming is based on semantic matching of posed queries with the stored semantics in the cache. Semantic matching demands efficient indexing scheme to the index semantics of already processed queries. Efficient indexing scheme significantly contributes towards faster query trimming algorithm.

Concept of cache has been used already as a mid-tier for database management systems to reduce the data latency [19–21]. However, cache used for database management system is a syntactic cache. Due to syntactic cache, partially overlapped queries cannot be answered from the cache; as a result data latency cannot be reduced up to the maximum extent. Partially overlapped queries can be answered from the cache by using semantic cache as a mid-tier instead of syntactic cache. Due to having ability to answer partially overlapped queries from the cache; semantic cache reduced the data latency more than the syntactic cache [18]. This research study is emphasized on SELECT-PROJECT queries. Rest of relational operations like Join, Set Operation, Cartesian product and others can be solved by transforming them into SELECT-PROJECT. For example, if there is a Join or Cartesian product between two tables, then first we have to project from one table, then project from another table, and finally do selection on obtained results. Similarly, set operations can be transformed into SELECT-PROJECT operations. Hence, it became obvious from above example, that any complex or simple query is actually some variation of simple SELECT PROJECT operations, if our scheme deals with SELECT PROJECT correctly, then it will be capable enough to provide a solution for any other operations even in complex situations.

In this study, we focused on SELECT-PROJECT queries only for relational databases. In fact, the proposed solution can be applied to rest of the databases like, XML database [22], column database [23] in which, data retrieved in SELECT-PROJECT fashion, too.

In this study, the existing techniques of query processing and semantic indexing are analyzed on the basis of above qualities (Accuracy, Increased Data Availability, Reduced Network Traffic, and Reduced Data Access Latency). The in-depth analysis of literature has led us towards the identification of existing gap, which include, there exist are major limitations in existing schemes of semantic cache query processing. A query processing algorithm is proposed which discovers even the hidden semantics of query and hence maximizes the hit ratio. A scheme to index the semantics in the cache is also proposed, which reduces the query matching time significantly if the number of segments in cache increases. The proposed scheme has been implemented and results are presented. On the basis of the results, it is concluded that the proposed technique performs better in terms of reducing data access time, increasing data availability, producing the correct result, and reducing network traffic.

1.2 Problem Statement

Analysis of the state-of-the-art semantic cache system on some defined benchmark is not available. So, this study focuses on the analysis of the state-of-the-art cache systems against benchmark given in Equation 1.8, 1.9, 1.10 and 1.11:

$$\mathrm{Data}_{\mathrm{ReminderQuery}} \cup \mathrm{Data}_{\mathrm{ProbeQuery}} \implies \mathrm{Data}_{\mathrm{UserQuery}} \tag{1.8}$$

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{Cache}} \implies \varnothing \tag{1.9}$$

$$\text{Segment}_{i\text{-cache}} \cap \text{Segment}_{j\text{-cache}} \implies \varnothing \forall i, j \text{ whenever } i \neq j \tag{1.10}$$

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{ProbeQuery}} \implies \varnothing \tag{1.11}$$

From the analysis, it is concluded that there exist no system that satisfies all points of benchmark given above. So, propose a system which complies with the above benchmark.

1.3 Research Questions

In order to solve the problem stated in Section 1.2, we have to address the following research questions.

1. What should be the criterion to evaluate the existing semantic cache systems?
2. How existing system should be evaluated on the defined criterion?
3. What are the limitations in the existing systems?
4. How a semantic cache system should be designed to overcome the limitations in existing systems? (To improve hit ratio, reduce network traffic, enhance data reusability, and improve time efficiency).
5. Can we design the algorithms (with reduced complexity) to overcome the limitations of existing systems?
6. In which area the proposed system performs better than existing systems?

1.4 Research Methdology

In order to answer the research questions raised in Section 1.3, to conduct the analysis of the problem given in Section 1.2 and to develop a new scheme/algorithm, following research methodology is followed.

1. A detailed criterion is defined to evaluate the existing schemes.

   Some benchmarks [24, 25] exist to evaluate query processing for databases but does not work for semantic cache query processing. In this study, we have defined the criterion to evaluate the semantic caching query processing.

2. Evaluation of existing semantic cache query processing schemes for relational queries.

   (a) Evaluation of implication/satisfiability algorithms

   (b) Evaluation of semantic indexing schemes

   (c) Evaluation of query matching algorithms

   (d) Evaluation of query trimming algorithms

3. Proposed an architecture for the semantic cache system with the proper placement of the designed algorithm

4. Developed efficient and less complex algorithm to:

   (a) Perform implication/satisfiability

   (b) Perform query matching

   (c) Perform query trimming

5. Designed a scheme to store and index query semantics intelligently

6. Evaluated the correctness of the algorithms by using case studies.

7. Performed a complexity analysis of the algorithms and compare it with existing algorithms.

8. Implemented an experimental test bed to demonstrate the working as well as to get real results.

9. Analyzed and reported the results.

Notation	Description	Notation	Description
S	Segment on Cache	Q_C	Cache Query
Q_U	User posed query	M_C	Matching Column
Q_A	Set of Attributes in user query	NM_C	Non Matching Column
Q_P	Condition	N_{MCC}	Non Matching Column in QC
Q_R	Relation of user query	N_{MCU}	Non Matching Column in QU
A_q	Amending query	O_{PC}	Operator in predicate of Cache Query
R_q	Remainder query	O_{PU}	Operator in predicate of User Query
P_q	Probe query	D_{VC}	Data Value of Cache Predicate
P_A	Predicate attribute	D_{VU}	Data value of User Predicate
S_A	Attributes of segment	R_V	Relationship between Data Value
S_P	Predicate of segment	O_{OP}	Opposite operator in Nature
S_R	Relation of segment	K_A	Key attribute of segment
S_{SA}	Status of Attribute	$\longleftarrow$	Assignment Operator
C_A	Common Attributes	D_A	Difference Attributes
P_{AS}	Predicate Attributes of Cache Query	P_{AU}	Predicate Attributes of User Query

Table 1.1: Notations

1.5 Definitions and Notations

This section presents some definitions by using relational algebraic notations that are used in the thesis. Some notational symbols may be found in Table 1.1.

Definition 1.1. **User Query** (Q_U) will be represented by 5-tuple $\langle D, Q_A, Q_P, Q_R, P_A\rangle$ where D is the name of database, Q_A is a set of required attributes, Q_R is a relation, Q_P is a condition and P_A is set of attributes in predicate.

Definition 1.2. Given a database $D = \{R_i\}$ and its attributes set $A = \cup A_{R_i}$, $1 \leq i \leq n$, **Semantic Enabled Schema**, Q_C will be 6-tuple $\langle D, S_A, S_P, S_R, S_{SA}, C\rangle$ where D is the name of database, S_R is name of the relation, S_A is a set of attributes, S_{SA} is a status of attributes, S_P is predicate (condition) on which data has been retrieved and cached, and C is the reference of contents.

Definition 1.3. Given a user query Q_U $\langle D, Q_A, Q_P, Q_R\rangle$ and Q_C $\langle D, SA, SP, SR, SSA, C\rangle$; **Data Set, $\mathbf{D_U}$** and **$\mathbf{D_C}$** will be the retrieved rows in the execution of Q_U and Q_C respectively.

Definition 1.4. Given a user query Q_U and cached query Q_C, **Probe Query** (pq) will be $Q_U \cap Q_C$ and data set against pq will be $D_U \cap D_C$.

Definition 1.5. Given a user query Q_U and cached query Q_C, **Remainder Query** (rq) will be QU - QC and data set against rq will be $D_U - D_C$.

Definition 1.6. Given a user predicate Q_P and cached predicate S_P, **Predicate Implication** $(Q_P \longrightarrow S_P)$ holds if and only if $(Q_P \cap S_P) = \varnothing$.

Definition 1.7. Given a user predicate Q_P and cached predicate S_P, **Predicate Satisfiability** holds if and only if $Q_P \cap S_P \neq \varnothing$.

Definition 1.8. Given a user predicate Q_P and cached predicate S_P, **Predicate Unsatisfiability** holds if and only if $Q_P - S_P = Q_P$.

Definition 1.9. Given a user query Q_U and cached query Q_C, **Query Implications** $(Q_P \longrightarrow Q_C)$ holds if and only if $Q_A \subseteq S_A$ and $Q_P \longrightarrow S_P$.

Figure 1.5 would be helpful to understand the concept of query implication.

Definition 1.10. Given a user query Q_U and cached query Q_C, **Query Satisfiability** holds if and only if $Q_A \cap S_A \neq \varnothing$ and $Q_P \cap S_P \neq \varnothing$.

Again, Figure 1.5 would be helpful to understand the concept of query satisfiability.

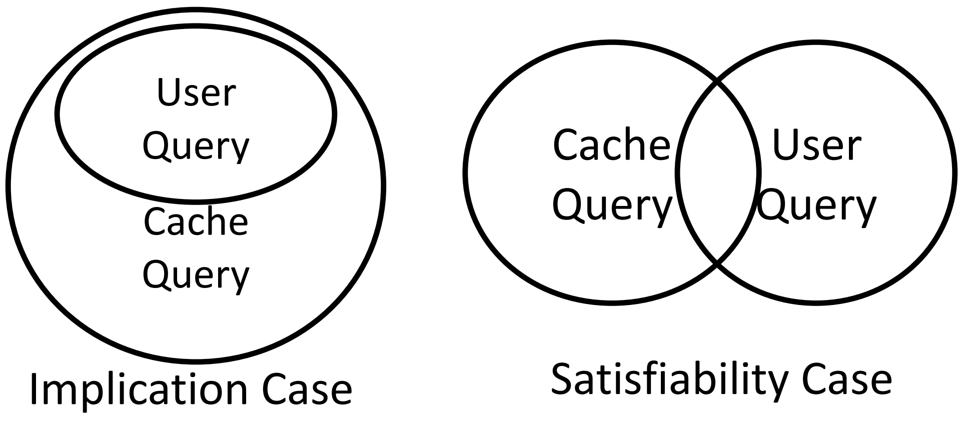

Figure 1.5: Satisfiability and Implication

Definition 1.11. Given a user query Q_U and cached query Q_C, **Query Unsatisfiaility** holds either when $Q_P \cap S_A = \varnothing$ or $Q_P \cap S_P = \varnothing$.

Definition 1.12. Given a user query Q_U and cached query Q_C, **Common Attributes** (C_A) is a set of attributes which are common among user and cached query and will be computed as $C_A = Q_A \cap S_A$.

Definition 1.13. Given a user query Q_U and cached query Q_C; **Difference Attributes** (D_A), the set of attributes which exists in user query, but not in cached query and will be computed as $D_A = Q_A - S_A$

S.No.	SR	SA	SP	SC
S_1	Student	age, name	age$<$ 28$\wedge$gpa$>$ 3.4$\vee$gpa$<$ 3.4	1
S_2	Student	age, city	age$>$ 28	2

Table 1.2: Segments on Cache

Definition 1.14. Given a database $D = \{R_i\}$ and its attributes set $A = \cup A_{R_i}$, $1 \leq i \leq n$, **Semantic Segment**, S is 4-tuple $\langle S_R, S_A, S_P, S_C \rangle$ where S_C is reference of contents, S_R is name of the relation, S_A is a set of attributes, and $S_p = P_1 \vee P_2 \vee \ldots \vee P_N$ where each P_i is a conjunction of simple predicates, i.e. $P_j = P_{j1} \wedge P_{j2} \wedge \ldots \wedge P_{jN}$ [18].

In order to understand the concept of a Semantic Segment in cache, the following example would be helpful.

Assume the cache has stored the data with semantics given in Table 1.2 and new query (Q_U) is posed by user as given below.

Q_U: *SELECT* city, gpa *FROM* student *WHERE* age$\geq$ 28

In the result of query trimming the above user query will be divided into the probe and the remainder query as follows.

pq: *SELECT* city, gpa *FROM* student *WHERE* age$>$ 28

rq:*SELECT* city, gpa *FROM* student *WHERE* age$=$ 28

Chapter 2

Criteria to Evaluate Existing Techniques

In Chapter 1, we have defined four qualitative measures as benchmark to evaluate the semantic cache system. These four qualitative points are generic. In order to quantify the evaluation of the semantic cache system, we have defined eight (six to measure four qualitative points of benchmarks and two qualitative points defined the scope of the system to evaluate) domain specific quantitative parameters in this chapter. These defined quantitative parameters would help to measure the four points of benchmark (Accuracy, Increased Data Availability, Reduced Network Traffic, and Reduced Data Access Latency) as illustrated in Chapter 1.

This chapter provides an answer for the first research question, listed in Section 1.3.

This chapter is further divided into two subsections, 2.1 and 2.2. In section 2.1, eight quantitative parameters are discussed in detail and Section 2.2 presents the association between six quantitative parameters and four points of benchmarks. In other words, section 2.2 presents discussion that how four qualitative points of benchmark can be ensured by using some of the six quantitative parameters.

2.1 Quantitative Parameters

This section presents the six quantitative parameters which would be helpful to measure the benchmark to evaluate the semantic cache system, defined in Chapter 1. These quantitative parameters are selected to ensure the accuracy of the semantic cache system, to increase the data availability, to reduce the network traffic, and to reduce the data access latency of the semantic cache system. Following are the eight quantitative parameters, first six are used to ensure the four qualitative points of benchmark and last two are used to define the scope of the semantic cache system.

- Semantics and Indexing Structure (IS)
- Generation of Amending Query (GoAQ)
- Zero Level Rejection (ZLR)
- Predicate Matching
- *SELECT_CLAUSE* Handling
- Complexity of Query Matching (CoQM)
- Query Type
- Predicate Formulae

Each quantitative parameter is discussed below, one by one.

Semantics and Indexing Structure

As discussed in Chapter 1 that semantic cache stores the data as well as semantics extracted from the query. In future, the semantics of a new user query are matched with the semantics of stored query in cache. This matching process really depends on two things. First, which type of semantics is stored and secondly the structure used to store the semantics in cache for previous queries. Unstructured storage of semantics in the cache will lead difficulty to write an algorithm to match the semantics of cached query with semantics of a new user query. Similarly, structured storage of semantics facilitates to write matching easily. The efficient indexing structure of semantics ensures efficient query matching algorithm.

Initially, semantics were indexed in a flat structure [10]. Query matching with flat structure proved costly therefore, semantics of queries were stored in segments to improve the efficiency of query matching [18]. Further improvement in query matching efficiency is achieved by 3-Level hierarchal indexing scheme [26, 27] and then by four-level scheme as 4-HiSIS [28]. In this research study, we proposed a schema based indexing scheme which is an enhancement of 4-HiSIS. So values for this parameter in the comparison table will be *Flat Structure, Segments, 3-Level Hierarchal, 4-HiSIS* and *schema based indexing*.

Generation of Amending Query

Idea of amending query is introduced in [18] with key contained segments. An amending query will be generated if projected attributes are present in the semantic cache, but a selection criterion of the query is not in the semantic cache. Following examples provide better overview to understand the concept of amending query.

Assume following query has been executed and the result is stored in the cache with its semantic description.

SELECT ename age *FROM* employee *WHERE* sal > 30000

Note that "sal" attribute is not stored in cache.

Now suppose that user poses a new query to process as follows;

SELECT ename age *FROM* employee *WHERE* sal > 35000

If we analyze the above scenario, all of data against new user query is available in cache, but attribute used in the selection criterion is not in the cache i.e. sal is not stored in cache. Due to which, data cannot be retrieved as query cannot be executed without the availability of 'sal' attribute in cache. In this situation, only key attribute is retrieved on given selection criterion is retrieved instead of all projected attributes. In this way, reusability of existing data is increased. This concept is useful in increasing the overall efficiency of the query processing [18]. Values for this parameter in the comparison table will be *Yes*, *No* or *N.A* (not applicable). Not applicable value is assigned in those cases where *SELECT CLAUSE* is not handled.

Concept of amending query may increase the cost in case of the composite key. In case of the composite key, if a user requires only one attribute, which is available in cache, but selection criteria is not available, then according to procedure; key attribute will be retrieved against selection criteria. In this case, two or more than two attributes will be retrieved instead of a single attribute.

Zero Level Rejection

Rejection of incorrect queries at the initial stage without further processing is referred as zero level rejection (ZLR). This parameter is defined to avoid the redundant processing and hence, to improve the overall query processing time. In some cases, users may pose the query with incorrect attributes (attributes that are not part of particular relation), database name, invalid condition or relation (relation that is not part of a particular database) name. For example, if user poses following incorrect queries to process:

SELECT ename age *FROM* emMloyee *WHERE* age>30
(relation not exist in database)
SELECT ename age *FROM* employee *WHERE* gpa>3.0
(gpa is not field of employee)
SELECT ename sal *FROM* student *WHERE* age>30
(sal is not field of student)

In these cases, the query should be rejected at zero level and should not be continued further because at the end, there would not be any retrieved result due to incorrect queries. Time can be saved by avoiding the useless processing of incorrect queries. Rejection of incorrect queries at the initial stage will increase the efficiency of the query processing [17] by stopping useless processing. In the comparison table, the value for this parameter is "*Yes*" or "*No*".

Predicate Matching

Predicate matching is processed to determine the rows available at cache (selection criterion for probe query) and not available in the cache (selection criterion for remainder query). In the result of predicate matching, it is decided which rows should be retrieved from the server and which rows should be retrieved from the cache. In order to understand the concept of predicate matching, following example could be helpful.

Assume following cached query Q_C and user query Q_U.

$$Q_C = SELECT\text{ename}\,WHERE\ (\text{age}>40)$$
$$Q_U = SELECT\text{ename}\,WHERE(\text{age}<50)$$

In above queries, S_P (the predicate of cache query) is age>40 and Q_P (the predicate of user query) is age<50.

In predicate matching process, matching between SP and QP will be performed and predicate for the remainder and probe query will be decided. If computed predicate for probe query is not NULL, then the cache will hit otherwise cache will miss.

In this example, predicate for probe query will be "age$<$ 50 and age$>$ 40" (cache hit) and predicate for the remainder query will be "age$\leq$ 40".

This process should be accurate to produce the correct result and should be quick to reduce the overall query processing time. The predicate matching process must be intelligent enough so that, it must ensure the reusability of the available data in the cache [17]. In another study [18], predicate matching is based on the implication and satisfiability [29, 30] and also some dynamic rule sets [27] were defined for predicate matching. Predicate attribute (PA) and predicate data value (PDV) are used in these dynamic rules (Sumalatha et al., 2007). Due to inefficiencies in implication/satisfiability and dynamic rule sets, required hit-ratio cannot be achieved [31]. The example below provides better overview to understand the inefficient predicate matching.

Here, we assume a user query (Q_U) and a segment on cache (S) predicates for the same elements are given as below:

$$Q_C = SELECT \text{ename} WHERE \text{ (age}> 40 \wedge \text{ Sal}> 50000)$$
$$Q_U = SELECT \text{ename} WHERE \text{ (age}< 50 \wedge \text{ Sal}> 50000)$$

In this example, the cache will hit and probe and remainder queries will be generated according to defined dynamic rule set and implication/satisfiability.

Meanwhile, if users pose queries having different attributes in selection criteria (WHERE CLAUSE) like:

$$Q_U = SELECT \text{ename} WHERE \text{ Name} = \text{'Komal'}$$

Then defined dynamic rule set and implication/satisfiability determines that predicate attribute of both (user query and cached query) are different. On the basis of this result, cache will not hit and probe query will be determined as NULL. Even some of the data may exist in the cache with following predicate:

$(\text{age} > 40 \wedge \text{Sal} > 50k) \wedge (\text{eName} = \text{‘Komal’})$

From the above example, it can be concluded that dynamic rules and implication/satisfiability is not able to achieve the required hit ratio. Hence, there should be a mechanism to generate probe query in these types of cases to increase the hit ratio. In the comparison table, the values for this parameter are complexities of each algorithm used to match predicate.

SELECT ALL Handling

In Select queries, select clause is important. This parameter will reduce the network traffic, and improve the response time. There is an important issue in the matching of *SELECTCLAUSE* "*" with selected attributes. For example;

Q_C: *SELECT* ename age *FROM* employee *WHERE* age> 30
(Cached Query)
Q_U: *SELECT* * *FROM* employee *WHERE* age> 30
(User posed query)

Now, the question crops up that how "*" of user query and "ename & age" of cached query will be matched? The efficient handling of *SELECT* "*" increases the hit ratio [17]. As best of our knowledge there exist no cache system in the literature, that deals with matching SELECT "*" of Q_U with stored semantics in the cache. In the comparison table, the values for this parameter will be *Yes*, *No*, or *N.A* (not applicable). Not applicable value is assigned in those cases where *SELECT CLAUSE* is not handled.

Query Type

This parameter defines the scope of the particular semantic cache system. Query type mentions that which type of query is handled. Values against this parameter are *SELECT* and *PROJECT*, *TOP*, *JOIN*, etc.

Complexity of Query Processing

This parameter defines the overall query processing complexity for particular semantic caching system. Complexity is computed, if an algorithm is given in a particular technique. Only two techniques we have found which consists of an algorithm for query processing over the semantic cache.

Predicate Formulae

This parameter mentions that which type of a predicate formula (*WHERE CLAUSE*) is handled. To get the answers from cache, semantic views have to be found as similar, equivalent or supersets of the query. Values across this parameter are *conjunctive* (C), *disjunctive* (D), *all formulas* (A) and *simple* (without any disjunct and conjunct) predicate (S)

2.2 Association between Benchmark and Parameters

This section describes how and which quantitative parameter ensures which point of qualitative benchmark. Accuracy, Availability, Network traffic, and access latency can be ensured by six domain specific parameters as discussed below.

Accuracy: Accuracy can be ensured by

- Predicate Matching
- Semantics and Indexing Structure

The accurate predicate matching algorithm ensures accurate cache system. The inaccurate predicate matching algorithm may lead to retrieve more or less data than required data against a query. In order to understand the concept of accurate predicate matching, following example would be helpful.

Q_C: *SELECT* ename *WHERE* age>40
Q_U: *SELECT WHERE* age<50

In the above example, if some predicate matching process generates age>40 for probe query and age<40 for remainder query then the retrieved result will be inaccurate. Hence, a predicate matching process should be accurate that ensures accurate semantic cache system.

Semantics and Indexing structure also ensures accuracy of the semantic cache system.

To generate accurate result for semantic cache system, correct and complete semantics should be stored. If some semantic indexing scheme stores incorrect descriptions of query in the cache, then it may generate incorrect results.

Increased Data Availability: Following domain specific parameters may help to increase the data availability for a semantic cache system.

- Semantics and Indexing Structure
- Amending Query
- *SELECT ALL* Handling

The semantic Indexing scheme may increase data availability by minimizing the redundant semantics in cache. Generation of amending query also increases the data availability because due to amending query maximum available data in cache is reused as discussed above. *SELECT ALL* also has the ability to maximize the utilization of available data in the cache.

Reduced Network Traffic: Network traffic can be reduced with the help of the following parameters.

- Zero Level Rejection
- Amending Query
- *SELECT ALL* Handling
- Semantic and Indexing Structures

Network traffic can be reduced by rejecting the incorrect queries at client side instead of sending the incorrect queries to the server which causes increased traffic in the network. Amending query is another parameter to reduce the network traffic due to retrieving only the key attributes from the server in amending query instead of all the required attributes. *SELECT ALL* handling maximizes the utilization of available data and least amount of data is retrieved from the server which reduces the network traffic. The efficient indexing structure also reduces network traffic by maximizing the utilization of available data in the cache.

Reduced Data Access Latency: Data access time of the semantic cache system can be reduced by applying following parameters.

Qualitative Benchmark	Quantitative Parameters
Accuracy	Predicate Matching & Semantic and Indexing Structures
Increased Data Availability	Semantic and Indexing Structures, Generation of Amending Query, & *SELECT ALL* Handling
Decreased Network Traffic	Zero Level Rejection, Generation of Amending Query, *SELECT ALL* Handling, Semantic and Indexing Structures
Decreased Data Access Latency	Semantic and Indexing Structures, Predicate Matching, Complexity, Generation of Amending Query, Zero Level Rejection, & *SELECT ALL* Handling

Table 2.1: Association between Qualitative Benchmark and Quantitative Parameters

- Semantics and Indexing Structure
- Predicate Matching
- Complexity of Query Matching
- Amending Query
- Zero Level Rejection
- *SELECT ALL* Handling

The efficient semantic indexing structure reduces the complexity of query matching process significantly as the incoming query will have to be matched with stored semantics. Faster predicate matching ensures quick response time and reduced data access latency. If a semantic cache system facilitates zero level rejection, the response time will be reduced as the incorrect queries will be rejected at the local machine. Amending query and *SELECT ALL* handling maximizes the utilization of cached data which results in reduction in data access latency.

Table 2.1 summarizes the association between four points of qualitative benchmark and six domain specific quantitative parameters.

Chapter 3

Related Work

This chapter presents the contemporary state-of-the-art approaches related to semantic cache systems. Existing semantic cache systems designed for database are evaluated based on the defined quantitative parameters.

This chapter presents the solution to the second and third research questions (see Section 1.3), via evaluating the existing systems on defined criteria and discussing the limitations in the existing system.

Existing semantic cache systems have been reviewed by Kumar *et al.* [32], too. Kumar *et al.* have presented only the summary of each of the existing semantic cache system. Existing semantic cache systems are not evaluated on any defined criteria due to which, it is difficult to select the appropriate system to implement the semantic cache. In fact, there is a need to evaluate the existing systems on some defined criteria which results in expressing the strengths and limitations of the systems. In this research study, we are aimed at evaluating the existing systems upon some defined criteria.

Semantic cache is used in a variety of platforms, such as, on the Web [33], distributed environment [34], in mobile applications [35], and in location based databases [36]. Sync kit has been presented as a toolkit by Benson *et al.* [37]. This toolkit has been designed for web browsers to enhance the performance of client-side by caching the data. Another technique has been introduced [38] to store the paths of query in the cache and enhance the efficiency of web systems by finding sub paths from the saved paths in cache. To enhance the efficiency of search engines, a cache based technique has been described by Cambazoglu *et al.* [39]. To cache the dynamic web contents, Soundararajan and Amza have designed a technique that enhances the performance of client-side [40]. To answer the XML queries over semantic cache work is done by Sumalatha *et al.* [41], Chen *et al.* [42], and Sanaullah *et al.* [43] Lui *et al.* have presented a cache based method is presented for mobile applications to enhance the efficiency in offline mode [44]. In this research study, we evaluated techniques for client-server databases one be one on the basis of the defined criteria. However, the techniques on web and mobile applications are beyond the scope of this study.

3.1 Godfrey's Scheme

A technique [16] on query processing over semantic cache is presented by Godfrey and Gryz. In this technique authors discuss the SELECT and PROJECT queries. There is no discussion about SELECT "*" type queries and generation of amending. They have not discussed the rejection of incorrect queries at zero level. No algorithm is presented for query processing. Query processing is discussed theoretically / given in plain English. Query indexing is used to make results more efficient. Query trimming policy is described for generating probe and remainder query. Flat structure is used to index the semantics. There are some limitations in this scheme [16], as listed below.

- Query matching is performed at the cache level, which is an expensive technique.
- No algorithm is provided for predicate matching.
- There is no amending query generation in the query processing system.
- The scheme is not able to reject an incorrect query at initial level.

3.2 Ren's Scheme

Another technique on query processing over the semantic cache with detailed algorithms is presented by Ren *et al.* [18]. In this technique, an algorithm is presented to process *SELECT* and *PROJECT* queries over the semantic cache. The main concept of this technique is the generation of amending query with key-contained segments to increase the hit ratio. I/S (implication/satisfiability) algorithms are used to match the predicate. They claimed that their proposed algorithm is capable to handle the disjunction of conjunction formulae, but referred papers where from they borrowed implication/satisfiability [29]. Authors of [45] claimed that the algorithms are able to handle only conjunctive formulas. Cache is divided into segments for indexing the semantics of previously executed queries. Strong concept of this technique is generation of amending query via key-contained segments. Generation of amending query made the technique efficient [17, 18]. By amending query, stored data can be reused efficiently. The algorithm in this technique is not capable of handling the *SELECT* "*" type queries and there is no way to reject incorrect queries at zero level. Query matching algorithm of this technique has following flaws:

A. Implication of Satisfiability In the given algorithm, query matching procedure is based on implication/satisfiability (Sun et al., 1989), (Chen and Roussopoulos., 1994) to match the predicate of query. The implication and satisfiability algorithm has the following problems:

1. In the reference [29, 45] it is clearly mentioned that these algorithms work only for conjunctive formulae, but this technique [18] claims that the query processing algorithm will work for the disjunction of conjunctive predicates. So, there is a clear contradiction in both studies. If referred algorithms are not able to handle the disjunctive expressions [29] then how would query matching algorithm work according to the claim of Ren *et al.* [18]?

2. Referenced algorithms for implication/satisfiability assume that variables of segment predicate must be included in the user's predicate. i.e. $P_{AS} = P_{AU}$, where, P_{AS} is a set of variables of segment predicate and P_{AU} is a set of variables of user's predicate. In other words, we can say that if predicate attributes of both user and cached segment are same then satisfiability will return *YES* otherwise it will return *NO*.

Example

$$Q_C = SELECT \text{ename} \, WHERE \ (\text{age}> 40 \wedge \text{ Sal}> 50000)$$
$$Q_U = SELECT \text{ename} \, WHERE \ (\text{age}< 50 \wedge \text{ Sal}> 50000)$$

In this example, algorithm will return *YES.*

Meanwhile, if users pose queries with different attributes like:

$$Q_U = \mathit{SELECT}\,\text{ENAME}\,\mathit{WHERE}\ \text{EName} = \text{“Komal”}$$

Then algorithm will return *NO* because $P_{AS} \neq P_{AU}$. Even some of the data exist in the cache with following predicate: (age$>$ 40$\wedge$ Sal$>$ 50k) $\wedge$ (Ename = Komal). From this example, we can say that this technique is not able to achieve required hit ratio.

B. Useless Processing In the given query matching algorithm, there is a chance of useless processing, especially when there is no data available in the cache. It can be explained clearly with the following example.

Example We assume that there is a segment 'S' in cache with following query.

$$S = \mathit{SELECT}\ \text{ename, Salary}\ \mathit{FROM}\ \text{employee}\ \mathit{WHERE}\ \text{age} > 30$$
$$S_A = \{\text{ename, Salary}\},\ S_P = \text{age} > 30,\ S_R = \text{employee}$$

Now, user poses following query 'Q_U'

$$Q_U = \mathit{SELECT}\ \text{age}\ \mathit{FROM}\ \text{employee}\ \mathit{WHERE}\ \text{age} > 30$$
$$Q_A = \{\text{age}\},\ Q_P = \text{age} > 30,\ Q_R = \text{employee}$$

According to case 3 of query matching algorithm [18] and based on above example, Q_A is a not a subset of S_A, so true result will be evaluated on the basis of this condition. Then implication will be checked. Result of implication will also be true because conditions (predicate) of both (Q_P and S_P) are same. Hence, the query will be divided into the probe and remainder query. At the end, there will be no result across probe query. So, there is a useless processing and wastage of resources. Even a single attribute does not exist, then how processing is done? If there would be n segments on cache, then same process would be executed n times, which would increase the run time complexity.

In these situations, a query should not proceed after checking the first condition that declares that there are no common attributes of posed query and cached segment.

C. Inefficient Generation of Amending Query In the given algorithm, a strong concept of the generation of amending query is proposed with the help of key-contained segments, but this idea is not handled efficiently in the query matching algorithm of [18].

Example

Suppose there is a segment on the cache as given below.

$$Q_C = \mathit{SELECT}\ \text{Salary}\ \mathit{FROM}\ \text{employee}\ \mathit{WHERE}\ \text{age} > 30$$
$$S_A = \{\text{ename, Salary}\},\ S_P = \text{age} > 30,\ S_R = \text{employee}$$

Now, user poses following query 'Q_U'

$$Q_U = \mathit{SELECT}\ \text{ename, post}\ \mathit{FROM}\ \text{employee}\ \mathit{WHERE}\ \text{age} > 30$$
$$Q_A = \{\text{ename}\},\ Q_P = \text{age} > 30,\ Q_R = \text{employee}$$

As conditioned attribute (Age) is not present in the segment, then how the result would be retrieved against probe query as given in case 3 of query matching algorithm without any amending query generated in this case? To improve the hit ratio, an efficient algorithm has been designed [31] to generate amending query.

3.3 Wan's Scheme

A technique on semantic caching has been presented by Wan and colleagues [46]. In this technique authors also discussed the query processing of *SELECT* queries. *SELECT* "*" type queries and generation of amending query is not applicable for this technique because *PROJECT* queries are beyond this technique. There is no defined way to reject incorrect queries at zero level. An algorithm has been presented to generate probe and remainder query on the basis of implication/satisfiability. The main focus of the technique is managing the cache contents. User posed query is matched with stored segments.

There are some weak points in this technique [46], which are listed below.

- Query matching is performed at the segment level, which is an expensive technique.
- Predicate matching is based on the implication and satisfiability, which is not comprised of disjunctive queries.

It is not possible to reject incorrect queries at initial level.

3.4 Cai's Scheme

A semantic cache system for aggregate queries [47] has been presented in by Cai and colleagues. An aggregate query is the composition of *SELECT* and *PROJECT* queries having some operation (count, max, min, etc). In this research activity, we consider that how *SELECT* and *PROJECT* portion of query is handled in this technique. *SELECT* "*" type queries and generation of amending query are not handled properly. There is no defined way to reject incorrect queries at zero level. An algorithm is provided to generate probe and remainder query on the basis of implication/satisfiability.

There are some weak points in this technique [47], which are listed below.

- An amending query is not generated. Due to this stored data on cache cannot be reused efficiently. Predicate matching is based on implication/satisfiability, which is not comprised of disjunctive queries.

3.5 Jonsson's Scheme

A technique on query processing over semantic cache has been presented by Jonsson *et al.* [48]. The focus of this technique is *SELECT* (*WHERECLAUSE*) queries and projections of queries are considered as static. *SELECT* "*" type queries and generation of amending query are not applicable to this technique, because *PROJECT* queries do not lie under the scope of this technique. There is no defined way to reject incorrect queries at zero level.

No specific rule is used to match the predicate in this technique. Probe and remainder predicates are generated simply by taking conjunct of user's and semantic predicates. Semantics of previous queries are stored in a flat structure, which proved costly at the time of query matching [18].

There are some weak points in [48], which are listed below.

- It is expensive due to the flat structure indexing scheme.
- It only works for *SELECT* queries (*WHERECLAUSE*) and makes the *PROJECT* queries (*SELECTCLAUSE*) static.

- It has no ability to reject the query at initial level.

3.6 Bashir's Scheme

Bashir and Qadir presented a technique [49] on query processing over the semantic cache. The focus of this technique is on the *SELECT* (*WHERECLAUSE*) queries and projection of queries is considered static (projected attributes for user and cached queries are same). In this technique, 112 possible scenarios have been described on the basis of six basic operators $(<, \leq, >, \geq, =, \neq)$. On the basis of 112 possible scenarios, a predicate matching algorithm is designed, which is able to compare simple predicates [49]. A hierarchal (4-HiSIS) approach [28] is used for indexing the semantics. According to 4-HiSIS posed query is matched with cache in four hierarchal steps. *SELECT* "*" type queries and generation of amending query are not applicable for this technique, because *PROJECT* queries are not included in the scope of this technique. There is no way to reject incorrect queries at zero level. There are some weak points in [49], which are listed below.

- Predicate matching rules are defined only for simple queries. There is no defined strategy for matching conjunctive and disjunctive predicates.
- It only works for *SELECT* queries (*WHERECLAUSE*) and makes the *PROJECT* queries (*SELECTCLAUSE*) static.
- It is not capable of rejecting the incorrect queries at initial level.

3.7 Sumalatha's Scheme

M. R. Sumalatha et al. presented a technique [26, 27] on query processing over the semantic cache. The focus of this technique is *SELECT* and *PROJECT* (SELECT/WHERECLAUSE) queries. Given algorithm in this technique is not capable of handling the *SELECT* "*" type queries as well as there is no method of rejecting incorrect queries at zero level.

PDV (predicate data value) is used to match the predicate in this technique. There is no defined way to generate an amending query in the absence of conditioned attribute. A 3-level hierarchal approach has been proposed to index the semantics of previous queries (Sumalathal et al., 2007a), (Sumalathal et al., 2007) [26, 27]. There are some weak points in [26, 27], which are listed below:

- The PV; predicate matching is not efficient because it is unable to achieve required hit ratio.
- No clear rule has been defined to make the predicates of probe and remainder queries.
- It is not capable of generating amending query, which increases the runtime complexity.
- It is not capable to reject incorrect queries at initial level.

3.8 Ehlers's Scheme

Ehlers and Freitag presented a technique [50] to process a query that retrieves top rows by using concept to top-k semantic caching scheme. The Initial concept of cache for top-k query was introduced by Xie *et al.* [51] but this concept was not for semantic cache. Ehlers introduced this concept for semantic cache. In this technique, top-k queries are processed by saving semantics of already processed queries. Concept of segments is used to store the segments. So the complexity of query matching for this technique is exponential. Presented technique is not able to handle *SELECT* * type queries as well as there is no idea of amending query in this technique. Due to these two deficiencies, hit ratio is not up to the mark. In this technique, satisfiablity/implication algorithms are used which are expensive for query processing.

Table 3.1 presents the summary of all reviewed state-of-the-art techniques discussed so far. Table 3.1 concludes that Ren et al. presented complete system for query processing over semantic cache. Moreover, the system presented by Ren et al. has some limitations in the context of predicate matching and complexity of query matching.

	Parameters							
Techniques	Predicate Formula	GoAQ	ZLR	SELECT ALL Handling	Predicate Matching	Indexing Structure	CoQM	Query Type
Ren *et al.*	A	Yes	No	No	$O\left(n^{2376}+k\right)$	Segments	$O\left(m \times n \times 2^n + \left(n^{2376}+k\right)\right)$	*SELECT* and *PROJECT*
Bashir and Qadir	S	N.A	No	N.A	$O\left(n^*k\right)$	4-HiSIS	N.A	*SELECT* and *PROJECT*
Sumalatha *et al.*	A	N.A	No	N.A	$O\left(n^{2376}+k\right)$	3-level hierarchial	N.A	*SELECT* and *PROJECT*
Jonsson *et al.*	A	N.A	No	N.A	NR	Segments	N.A	*SELECT*
Godfrey and Gryz	A	No	No	No	NR	Flat Structure	N.A	*SELECT* and *PROJECT*
Cai *et al.*	C	No	No	No	$O\left(n^{2376}+k\right)$	3-level hierarchial	N.A	*SELECT* and *PROJECT*
Wan *et al.*	A	N.A	No	N.A	$O\left(n^{2376}+k\right)$	Segments	N.A	*SELECT* and *PROJECT*
Ehlers and Freitag	A	No	No	No	$O\left(n^{2376}+k\right)$	Segments	$O\left(m \times n \times 2^n + \left(n^{2376}+k\right)\right)$	Top-k

Table 3.1: Analysis Matrix of Existing Techniques

Chapter 4

sCacheQP

In the previous chapter, we have discussed the scholarly works of various researchers. By summing up the last chapter; we evaluated the existing semantic cache systems against the parameters as, Indexing Structure (IS), Generation of Amending Query (GoAQ). Zero Level Rejection (ZLR), Predicate Matching, *SELECT_CLAUSE* Handling, Complexity of Query Matching (CoQM), Query Type, and Predicate Formulae. In this chapter, with the help of algorithms and diagrams, we will discuss the proposed semantic cache system which overcomes limitations of existing systems as discussed in previous chapters.

This chapter provides answers of research questions 4 and 5 listed in Section 1.3 by developing algorithms (query matching and query trimming) and indexing (store semantics) scheme that overcomes the limitations of existing semantic cache systems.

Proposed Semantic Cache System: sCacheQP This section presents the sCacheQP, which is a complete architecture of the query processing system. Complete working of proposed system is depicted in Figure 4.1. We have divided the working of the system in four major building blocks; Query Matching, Query Trimming, Rebuilder, and Semantic Cache (which elaborates schema based indexing scheme).

There are four major building blocks in the proposed architecture. Detailed description of Query matching is presented in Section 4.1, Query trimming is presented in Section 4.2, rebuilder is further explained in Section 4.3, and schema

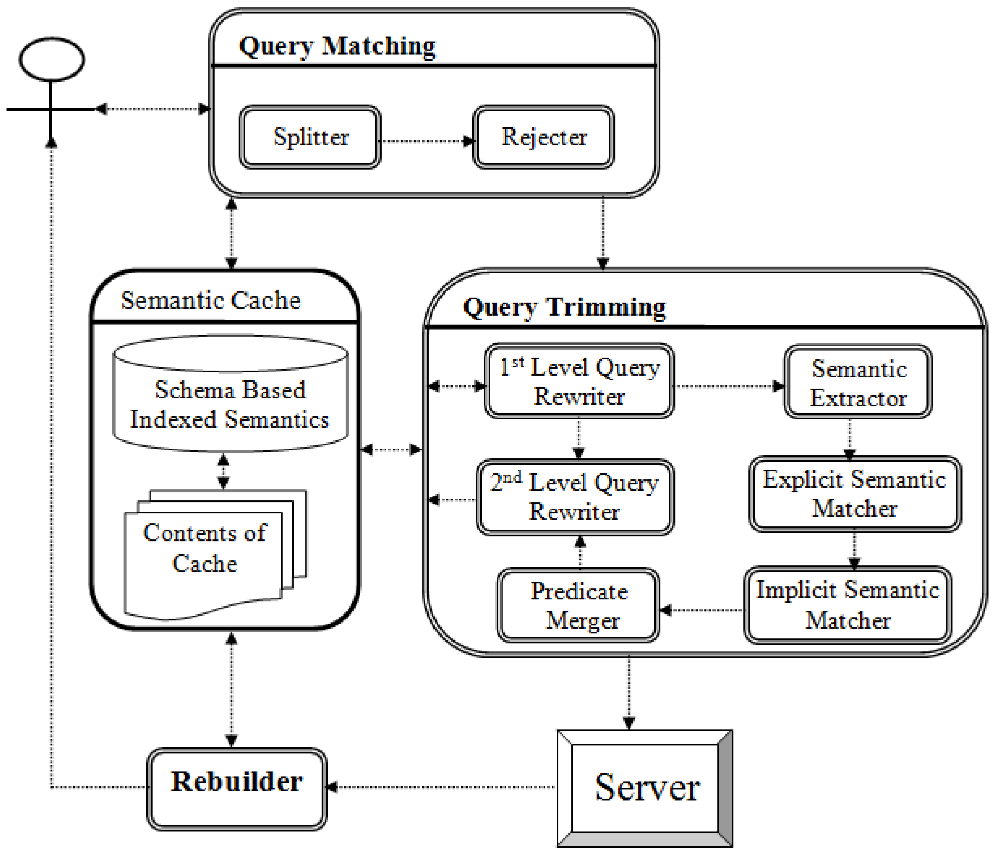

FIGURE 4.1: Semantic Cache Architecture

based indexing scheme is elaborated in Section 4.4. The basic driver algorithm is given in Algorithm 1.

Algorithm 1 accepts user posed query and cached query and returns the final result of the user query. At Step 4, variables are initialized. In the next, Split_Query(Q_U) algorithm is called, which accepts user query and returns different portions of the query. Working of Split Query is given in Algorithm 2. In Step 6, validity of the query is checked. If user posts a valid query, then this algorithm returns false, otherwise returns true. If returned result is false then sCacheQP moves on Step 12 otherwise it moves on Step 30. In Step 12, common and different attributes are computed. If set of difference attributes is empty, then Step 18 will be skipped otherwise rq_1 will be computed. In case of emptiness of common attributes set there will be no probe query, otherwise probe query and rq_2 will be computed in Steps 24 to 28. Working of each step is given next steps along with their algorithms. Working of each building block of sCacheQP with their algorithm is presented in further subsections.

Algorithm 1 Running Algorithm for sCacheQP

```
1:  Inputs:
        $Q_U, Q_C$
2:  Outputs:
        $F_R$ (Result against $Q_U$)
3:  Procedure:
4:  Initialize:
        $pq \leftarrow$NULL
        $rq_1 \leftarrow$NULL
        $rq_2 \leftarrow$NULL
5:  $PORTIONS \leftarrow$ $SPLIT\_QUERY$ $(Q_U)$;
6:  $Reject \leftarrow$ $CHECK\_REJECTION$ $(PORTIONS)$;
7:  if Reject = false then
8:      goto Line 12;
9:  else
10:     Reject Query and goto Line 30;
11: end if
12: $C_A, D_A \leftarrow$ 1st_Level_Query_Rewriter $(QA, SA)$;
13: if $D_A \neq$ empty then
14:     goto Line 18;
15: else
16:     goto Line 19;
17: end if
18: $rq_1 \leftarrow \pi D_A \sigma Q_P$ $(Q_R)$ ;
19: if $C_A \neq$empty then
20:     goto Line 24;
21: else
22:     goto Line 29;
23: end if
24: $M_C, N_{MCC}, N_{MCU}, Dvc, Dvu, Opc, Opu, Coc, Cou \leftarrow$Semantic_Extraction$(Q_P, S_P)$;
25: $C_1, N_{C1} \leftarrow$ExplicitySemanticMatching$(M_C, Dvc, Dvu, Opc, Opu)$;
26: $C_2, N_{C2} \leftarrow$ImplicitSemanticMatching$(M_C, N_{MCC}, N_{MCU}, C_1, N_{C1})$
27: Cached, $N$-Cached$\leftarrow$PredicateMerging $(C_2, N_{C2}, Coc, Cou)$
28: $pq, rq_2 \leftarrow$ 2nd_Level_Query_Rewriter $(Q_F, C_A,$ Cached, $N$-Cached$)$;
29: $aq \leftarrow G_{EN}\_AMEND\_QUERY$ ();
30: $F_R \leftarrow$ Rebuilder$(pq, rq_1, rq_2)$;
```

4.1 Query Matching

In semantic cache, user posed query is matched with the stored semantics on the cache. Through this process, the decision is taken whether the data is available in the cache or not. Query matching process is accomplished in two sub processes, splitter and rejecter. Splitter accepts the user query Q_U from the user interface and splits the query on the basis of three clauses of the query *SELECT*, *FROM*, and *WHERE*. These three portions are called Q_A (*SELECT: projected attributes in the user query*), Q_R (*FROM*: Relation) and Q_P (*WHERE*: selected rows/tuples on specific conditions); and sends to the rejecter for initial level checking. Q_P will be empty, if there is no condition on user posed query [31, 52]. An Algorithm for splitting the user query was presented by Ahmad *et al.* and is given in Algorithm 2.

Algorithm 2 Split_Query()

```
1: Input:
       $Q_U$
2: Outputs:
       $Q_A, Q_P, Q_R$
3: Procedure:
4: $Q_A \leftarrow$ SELECT CLAUSE;
5: $Q_P \leftarrow$ WHERE CLAUSE;
6: $Q_R \leftarrow$ FROM CLAUSE;
7: Return $\leftarrow Q_A, Q_P, Q_R$;
```

Responsibility of rejecter is to check the validity of user posed query by sending the list of selected attributes (Q_A), relation (Q_P) and predicate attributes (P_A) on the schema based indexing semantics. Predicate attribute is extracted by rejecter from Q_P and included in the list. If attributes list of Q_A, Q_R and P_A matches the stored schema, then processing would continue, otherwise query would be rejected and processing would stop. Rejecter also builds Q_A in the case of '*' by retrieving all attributes from schema as a list if predicate attribute exist in schema [31]. Algorithm to validate the user query was presented by Ahmad et al. [31] and is given in Algorithm 3. This algorithm may be helpful to decrease the network traffic as well as to reduce the data access latency by rejecting the incorrect queries locally (no need to send request to the server), which leads to a reduction of the network traffic and data access latency.

Algorithm 3 Check_Rejection

```
1: Inputs:
       $Q_A, Q_P, Q_R$                                        $Q_U$
2: Output:
       True/False
3: Procedure:
4: if all attributes of $Q_A$ present in schema then
5:     if relation of $Q_R$ present in schema then
6:         if $P_A$ is present in schema then
7:             if $Q_A = ^*$ then
8:                 Return false and build $Q_A$ from schema;
9:             else
10:                Return true;
11:            end if
12:        else
13:            Return true;
14:        end if
15:    else
16:        Return true;
17:    end if
18: else
19:    Return true;
20: end if
```

4.2 Query Trimming

When it has been decided that data is available in the cache, then second step of sCacheQP is performed. In this step query is divided into two sub queries called probe and remainder queries, this process is named as query trimming. This process is accomplished in two stages. At first stage, vertical partition takes place and the attributes that are not available (D_A) at cache are directly sent to the server as rq_1 (remainder query) with original predicate.

We called it 1^{st} level query rewriter [31] and its algorithm is given in Algorithm 4. The query rq_1 is computed as follows:

$$\boxed{rq_1 = \pi_{DA}\sigma_{QP}\left(Q_R\right)}$$

Rest of attributes; that are common in both user and cached query, are forwarded to the predicate processor, which works on the 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

Algorithm 4 $1^{st}Level\_Query\_Rewriter\ (Q_A, S_A)$

```
1: Inputs:
       $Q_A, S_A$
2: Outputs:
       $C_A, D_A$
3: Procedure:
4: $C_A \leftarrow Q_A \cap S_A$ ;
5: $D_P \leftarrow Q_A - S_A$ ;
6: Return $\leftarrow C_A, D_A$;
```

simplified by just separating the portions of it on the basis of conjunctive and disjunctive operators. Then semantics of user's query predicate with respect to the cached predicate is extracted in the form of matching columns (M_C – similar in both user query predicate and cached predicate), non-matching columns of cache (NM_C – columns in cached query that do not match with user query) and non-matching columns of user query (NM_u – columns in user query that do not match with cached query). Some other information is also collected like; data values of the cache predicate (D_{VC}), data values of the user predicate (D_{VU}), comparison operators in the cache predicate (*Opc*), comparison operators in the user predicate (*Opu*). Algorithm to extract the semantics of predicate is given below in Algorithm 5.

Algorithm 5 Semantics Extractor

```
1: Inputs:
       $Q_P, S_P$
2: Outputs:
       $Coc[n], Cou[n], M_C[n], N_{MCC}[n], N_{MCU}[n], Opc[n], Opu[n], Dvc[n], Dvu[n]$
3: Procedure:
4: $MC[n] \leftarrow$ List of Columns Present in both $Q_P, S_P$;
5: $N_{MCC}[n] \leftarrow$ List of Columns Present in $S_P$ but not in $Q_P$;
6: $N_{MCU}[n] \leftarrow$ List of columns present in $Q_P$ but not in $S_P$;
7: $Opc[n] \leftarrow$ operator set of $S_P$;
8: $Opu[n] \leftarrow$ Operator present set of $Q_P$;
9: $Dvc[n] \leftarrow$ Data values in $S_P$;
10: $Dvu[n] \leftarrow$ Data values in $Q_P$;
11: $Coc[n] \leftarrow$ Connective Operators in $S_P$;
12: $Cou[n] \leftarrow$ Connective Operators in $Q_P$;
13: Return $Coc[n], Cou[n], M_C[n], N_{MCC}[n], N_{MCU}[n], Opc[n], Opu[n], Dvc[n], Dvu[n]$;
```

After extraction of semantics $Mc, D_{VC}, D_{VU}, Opc, andOpu$ are sent to the Explicit Semantic Matcher. Explicit Semantic Matcher trims the predicate into two portions; one for the remainder ($C1$) and other for probe query (N_{C1}). The explicit

Semantic Matching algorithm is based on the boundary values as well as on the nature of the comparison operators instead of satisfiability/implication. There are 112 rules defined on the basis of boundary values and basic comparison operators ($<, \leq, >, \geq, ==$ and $\neq$). Rule based Algorithm is given in Algorithm 6, which is designed to match the explicit semantics of predicates. Designed algorithm divides the predicate into cached predicate (predicate for probe query) and non-cached predicate (predicate for remainder query). This algorithm proved to be helpful to ensure all the four points of benchmark as discussed below.

Algorithm 6 ExplicitSemanticMatching

```
1: Inputs:
     $Mc[n], O_{Pc}[n], O_{Pu}[n], D_{Vc}[n], D_{Vu}[n], C_C[ni], C_U[n]$
2: Outputs:
     $C_1[n], N_{C1}$
3: Procedure:
4: Initialize:
     $C_1[n]$ ←Null, $N_{C1}[n]$ ← Null
5: for $i = 0$ to $n$ do
6:   if $D_{V_C}[i] < D_{V_U}[i]$ then
7:     if $(O_{PC}[i] \in \{\neq, >, \geq\}) \wedge (O_{PU}[i] \in \{>, \geq, =\})$ then
8:       $C_1[i] \leftarrow C_C[i]\ O_{PC}[i]\ D_{V_C}[i]$
9:       $N_{C1}[i]$ ←Null;
10:    else if $(O_{PU}[i] \in \{<, \leq, \neq\})$ then
11:      $C_1[i] \leftarrow C_C[i] O_{P_C}[i] D_{V_C}[i]$
12:      $N_{C1}[i] \leftarrow (C_U[i]\ O_{PU} D_{V_U}[i]) \wedge (C_C[i] \text{Rev} (O_{Pc}[i])\ D_{Vc}[i])$
13:    else$C_1[i]$ ←Null  $N_{C1}[i] \leftarrow (C_U[i] O_{Pu}[i] D_{V_U}[i])$
14:    end if
15:  else if $D_{V_C}[i] > D_{V_U}[i]$ then
16:    if $(O_{PC}[i] \in \{\neq, <, \leq\}) \wedge (O_{PU}[i] \in \{<, \leq, =\})$ then
17:      $C_1[i] \leftarrow C_C[i] O_{P_C}[i] D_{V_C}[i]$;
18:      $N_{C1}[i]$ ←Null;
19:    else if $(O_{P_C}[i] \in \{\neq, >, =, <, \leq\}) \wedge (O_{P_U}[i] \in \{>, \geq, \neq\})$ then
20:      $C_1[i] \leftarrow C_C[i] O_{P_C}[i] D_{V_C}[i]$
21:      $N_{C1}[i] \leftarrow (C_U[i]\ O_{PU} D_{V_U}[i]) \wedge (C_C[i] \text{Rev} (O_{P_C}[i])\ D_{V_C}[i])$
22:    else
23:      $C_1[i]$ ←Null
24:      $N_{C_1}[i] \leftarrow (C_U[i] O_{P_U}[i] D_{V_U}[i])$
```

```
25:         end if
26:     else if $D_{V_C}[i] = D_{V_U}[i]$ then
27:         if $((O_{P_C}[i] \in \{\geq\}) \wedge (O_{P_U}[i] \in \{>, , =\}))$
28: $\vee (O_{P_C}[i] = O_{P_U}[i]) \vee ((O_{P_C}[i] \in \{\geq\})$
29: $\wedge (O_{P_U}[i] \in \{<, =\}) \vee ((O_{P_C}[i] \in \{\neq\}) \wedge (O_{P_U}[i] \in \{<, >\}))$ then
30:             $C_1[i] \leftarrow C_C[i]O_{P_C}[i]D_{V_C}[i]$;
31:             $N_{C1}[i] \leftarrow$Null;
32:         else if $(O_{P_C}[i] \in \{>, \leq\}) \wedge (O_{P_U}[i] \in \{\geq, \neq\})$
33: $\vee ((O_{P_C}[i] \in \{<, \geq\}) \wedge (O_{P_U}[i] \in \{\leq, \neq\}))$
34: $\vee (O_{P_C}[i] \in \{\neq, =\}) \wedge (O_{P_U}[i] \in \{\leq, \geq\})$ then
35:             $C_1[i] \leftarrow C_C[i]O_{P_C}[i]D_{V_C}[i]$
36:             $N_{C_1}[i] \leftarrow (C_U[i]\ O_{P_U}D_{V_U}[i]) \wedge (C_C[i]\mathrm{Rev}\,(O_{P_C}[i])\,D_{V_C}[i])$
37:         else
38:             $C_1[i] \leftarrow$Null
39:             $N_{C_1}[i] \leftarrow (C_U[i]O_{P_U}[i]D_{V_U}[i])$
40:         end if
41:     else if $(D_{V_C}[i] \neq D_{V_U}[i]) \wedge ((O_{P_C}[i] \in \{=, \neq\}) \wedge (O_{P_U}[i] \in \{\neq\}))$ then
42:         $C_1[i] \leftarrow C_C[i]O_{P_C}[i]D_{V_C}[i]$;
43:         $N_{C_1}[i] \leftarrow (C_U[i]\ O_{P_U}D_{V_U}[i]) \wedge (C_C[i]\mathrm{Rev}\,(O_{P_C}[i])\,D_{V_C}[i])$
44:     else
45:         $C_1[i] \leftarrow$Null
46:         $N_{C_1}[i] \leftarrow (C_U[i]O_{P_U}[i]D_{V_U}[i])$
47:     end if
48: end for
```

Accuracy is ensured because the algorithm divides the user query predicate into cached predicate and non-cached predicate accurately. The algorithm ensures two qualities while dividing the user query predicate; firstly, nothing is overlapped between cached queries and non-cached predicates. Secondly, nothing is included or excluded unnecessarily in resultant cached and non-cached predicates.

This algorithm maximizes the utilization available data in the semantic cache by dividing the user query predicate into cached predicate and non-cached predicate. Due to a maximum utilization of data, availability of the system is increased; less data retrieval from server causes the reduction in network traffic and data access latency.

The output of a predicate matching algorithm is a predicate that is available at cache (C_1) and predicate that is not available in the cache (N_{C1}). The working of the algorithm is explained below. As we have discussed that Explicit Semantic Matching algorithm is based on the boundary value and basic comparison operator. On the basis of boundary value and comparison operators; Algorithm 6 will trim the predicate into the probe and remainder queries.

Remember that predicate matching algorithm having better time complexity is an alternative of satisfiability/implication [30] used to help process query in the literature [18, 53]. Computed values C_1, N_{C1} and N_{MU} are sent to the Implicit Semantic Matching algorithm to remove the additional information. Algorithm for performing this job is given below in Algorithm 7.

Algorithm 7 Implicit Semantic Matching

```
1:  Inputs:
        $M_C[n], N_{MCC}[n], N_{MCU}[n], C_1[n], N_{Cq}[n]$
2:  Outputs:
        $C_2[n], N_{C2}$
3:  Procedure:
4:  Initialize:
        $C_2 \leftarrow$ Null, $N_{C_2} \leftarrow$null
5:  for $i = 0$ to $n$ do
6:      if $(N_{MCC}[i] = \text{null}) \wedge (N_{MCU}[i] = \text{null})$ then
7:          $C_2 \leftarrow C_1[i]$;
8:          $N_{C2} \leftarrow N_{C1}[i]$
9:      else if $(N_{MCC}[i] \neq \text{null}) \wedge (N_{MCU}[i] = \text{null})$ then
10:         $C_2 \leftarrow (C_1[i]) + (NMCC[i])$;
11:         $N_{C2} \leftarrow ((C_1[i] + \text{R}(N_{MCC}[i])) \vee (N_{C1}[i])$
12:     else if $(N_{MCC}[i] = \text{null}) \wedge (N_{MCU}[i] \neq \text{null})$ then
13:         $C_2 \leftarrow (C_1[i]) + (N_{MCU}[i])$;
14:         $N_{C2} \leftarrow N_{C1}[i] + N_{MCU}[i]$;
15:     else if $(N_{MCC}[i] \neq \text{null}) \wedge (N_{MCU}[i] \neq \text{null})$ then
16:         $C_2 \leftarrow (C_1[i]) + (N_{MCU}[i]) + (N_{M_C}[i])$;
17:         $N_{C2} \leftarrow ((C_1[i]) + \text{R}(N_{MCU})) \vee ((N_{C_1}) + (N_{MCU}[i]))$;
18:     else if $M_C[i] ==$ null then
19:         $C_2 \leftarrow (N_{MCC}[i]) + (N_{MCU}[i])$;
20:         $N_{C_2} \leftarrow N_{MCU}[i] + \text{R}(NMCC[i])$;
21:     end if
22: end for
```

Due to extraction of hidden semantics, this algorithm maximizes the utilization of available data which results in increased data availability, reduced network traffic, and reduced data access latency.

Generated cached (C_2) and non-cached (N_{C2}) predicates by the Implicit Semantic Matcher are combined by predicate merger. Algorithm to merge the predicate is given in Algorithm 11.

The computed predicates are then sent to the 2nd level query rewriter. Finally, probe and remainder queries are computed by the 2nd level query rewriter as given in Algorithm 8.

Algorithm 8 2^{nd}*Level_Query_Rewriter* (Q_A, S_A)

```
1: Inputs:
       $Q_A, C_A$,Cached, $N$-Cached
2: Outputs:
       $pq, rq_2$
3: Procedure:
4: $pq \leftarrow \pi_{C_A}\sigma_{\text{Cached}}(Q_R)$;
5: $rq_2 \leftarrow \pi_{C_A}\sigma_{N-\text{Cached}}(Q_R)$;
6: Return $pq, rq_2$
```

pq is executed locally and rq_2 is sent to the server. Then results of both are sent to the rebuilder to combine the result. If the predicate attribute of user query (P_A) is not available in the cache then amending query will also be generated as given in Algorithm 9. This algorithm may be helpful to maximize the utilization of available data in the cache, which leads to increased data availability, reduced network traffic, and reduced data access latency.

Algorithm 9 GEN AMEND QUERY

```
1: Inputs:
       $pq, S_P, Q_R, Q_P, P_A, K_A$
2: Output:
       $aq$
3: Procedure:
4: if $S_P \subseteq Q_P \| S_P = Q_P \| pq =$null $\| P_A \subseteq S_P$ then
5:     $aq \leftarrow$null;
6: else
7:     $aq \leftarrow \pi_K \sigma_{QP}(Q_P)$;
8: end if
```

4.3 Query Rebuilding

Rebuilder receives the result form server (S_R), which is retrieved, against the remainder queries ($rq1$ and rq_2) and results from cache (C_R) against the probe query (pq), combines both as a final result F_R. The final result is shown to the user and also updated in the cache contents, if required.

4.4 Schema Based Indexing Scheme

This section describes the schema based hierarchal indexing scheme for contents matching over the semantic cache. One of the challenges of query processing is an efficient query matching over the semantic cache. So to make the query processing efficient, there is a need to make the process of query matching accurate and effective. As discussed in Section 2, there are five different strategies adopted previously. 4-HiSIS (Bashir and Qadir., 2006) approach is more efficient than remaining schemes. This approach is used to find out whether required attributes are available from the cache or not. After finding the status of required attributes, this scheme also moves to segment based scheme for building probe and remainder queries, which is costly.

As we need a schema in semantic cache to process the query [17], we have enhanced indexing the scheme that will use schema to maintain the semantics of previous queries instead of storing semantics and schema separately. This research study introduces a way by which semantics of queries can be preserved efficiently by managing schema. This scheme is named as schema based indexing scheme for semantic cache and is able to reduce the query matching complexity into polynomial from exponential (Ahmad et al. 2016)[reference not found].

Initially schema for each database is stored in the cache with database names, relation names and attributes names with false status. Semantic enabled schema can be managed for any database. Here we have managed the schema of the library with semantics as given in Table 4.1.

Now suppose, that user enters a query as given below.

Database Name	**Relation Names**	**Attribute Names**	**Attribute Status**	**Condition**	**Content Reference**
Library	Books	Author	FALSE		
		Title	FALSE		
		ISBN	FALSE		
	Journals	Author	FALSE		
		Title	FALSE		

Table 4.1: Schema for Library

Database Name	**Relation Names**	**Attribute Names**	**Attribute Status**	**Condition**	**Content Reference**
Library	Books	Author	TRUE	Author=‘Ali’	1
		Title	TRUE	Author=‘Ali’	1
		ISBN	FALSE		
	Journals	Author	FALSE		
		Title	FALSE		

Table 4.2: Updated indexed schema

```
SELECT Author, Title FROM Books WHERE Author=‘Ali’
```

The stored contents will be updated as given in Table 4.2 on the basis of above queries.

Schema based semantics on cache are updated on the basis of a query. Author and Title across books are posed to retrieve the query result, so their status will be changed to true, and condition and content reference will also be updated.

After managing semantics; next step is to use the managed semantics on the cache. For content matching; first of all database names will be matched exactly. Secondly, if database names get matched, then relation names will be matched exactly otherwise, processing will be stopped. If relation names get matched, then attribute names will be matched. If required attributes are part of the schema then their status will be checked. If the status is true, then data across particular attribute will be available, otherwise it will be retrieved from the server. If the status of the attribute is also true, then condition across the attributes is matched. Finally, probe query is generated by generated conditions from referenced contents. A generic approach for schema based content matching is presented in Figure 4.2.

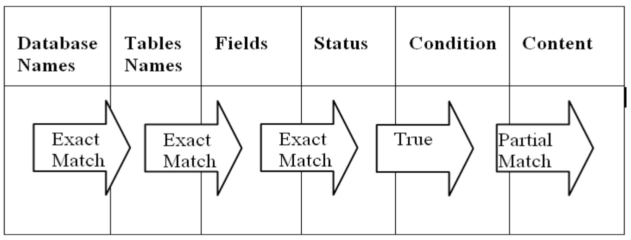

Figure 4.2: Schema based content matching

Proposed schema based indexing scheme is basically an enhancement of previous work 4-HiSIS [28].

There is a simple driving algorithm (sMatch) given to perform the query matching in Algorithm 10.

Algorithm 10 sMatch

```
1: Inputs:
       $D_A, Q_W, P, C_A$
2: Output:
       sMatch
3: Procedure:
4: if $D_A \neq \varnothing$ and $C_A \neq \varnothing$ then
5:     $rq_1 \leftarrow \pi_{D_A}\sigma_{Q_P}(Q_R)$ and $pq \leftarrow$NULL and $rq_2 \leftarrow$NULL;
6: else if $Q_W \rightarrow P$ and $D_A \neq \varnothing$ and $C_A \neq \varnothing$ then
7:     $rq_1 \leftarrow \pi_{D_A}\sigma_{Q_P}(Q_R)$ and $pq \leftarrow \pi_{C_A}\sigma_{Q_P}(Q_R)$ and $rq_2 \leftarrow$NULL;
8: else if $Q_W \rightarrow P$ and $D_A = \varnothing$ and $C_A \neq \varnothing$ then
9:     $rq_1 \leftarrow$NULL and $pq \leftarrow \pi_{C_A}\sigma_{Q_P}(Q_R)$ and $rq_2 \leftarrow$NULL;
10: else if $Q_W \wedge P$ is satisfiable and $C_A \neq \varnothing$ and $D_A \neq \varnothing$ then
11:     $rq_1 \leftarrow \pi_{D_A}\sigma_{Q_P}(Q_R)$ and $pq \leftarrow \pi_{C_A}\sigma_{Q_P}(Q_R)$ and $rq_2 \leftarrow \pi_{C_A}\sigma_{Q_P \neg P}(Q_R)$;
12: else if $Q_W \wedge P$ is satisfiable and $C_A \neq \varnothing$ and $D_A = \varnothing$ then
13:     $rq_1 \leftarrow$NULL and $pq \leftarrow \pi_{C_A}\sigma_{Q_P}(Q_R)$ and $rq_2 \leftarrow \pi_{C_A}\sigma_{Q_P \neg P}(Q_R)$;
```

Algorithm 11: Predicate Merging

```
1:  Inputs:
            $C_2[n], N_{C2}[n], Coc[n], Cou$
2:  Outputs:
            Cached, $N$-cached
3:  Procedure
4:  If $Coc \wedge Cou$ $\in$Null then
            Cached$\leftarrow C_2$;
6:          $N$-Cached$\leftarrow N_{C2}$;
7:  If Coc$\wedge$Cou$\in$ $\mathbf{\Lambda}$ then
8:          Cached$\leftarrow \wedge C_i$;
9:          $N$-Cached$\leftarrow \vee (C_i \wedge NC_j)$
                where $1 < i, j < n$ and $i \neq j$;
10: If Coc$\wedge$Cou$\in$ $\mathbf{V}$ then
11:         Cached$\leftarrow \vee C_i$ where $1 < i < n$;
12:         $N$-Cached$\leftarrow \vee NC_i$ where $1 < i < n$;
13: If Cou$\in$ $\mathbf{V}$ $\wedge$ Coc$\in$ $\mathbf{\Lambda}$ then
14:         Cached$\leftarrow \wedge C_i$;
15:         $N$-Cached$\leftarrow \vee (C_i \wedge NC_j)$
            where $1 < i, j < n$ and $i \neq j$;
16: If Cou$\in$ $\mathbf{V}$ $\wedge$ Coc$\in$ $\mathbf{\Lambda}$ then
17:         Cached$\leftarrow \wedge C_i$;
18:         $N$-Cached$\leftarrow \vee (C_i \wedge NC_i) \vee (U_i \wedge R(U_j))$ where $1 < i, j < n$ and $i \neq j$;
```

Proposed schema based hierarchical semantic indexing scheme has following advantages:

- Minimize the redundant semantics on a cache, which results in increased data availability.

- Query matching becomes faster and accurate, which leads to reduced data access latency and accuracy of the system.
- Rejects incorrect queries due to having schema at the cache, which achieves reduced network traffic and reduced data access latency.
- Handles SELECT * type queries, which achieves reduced network traffic and reduced data access latency.
- Reduce overall complexity of query processing for semantic cache which results in reduced data access latency.

Chapter 5

Analysis of Proposed System

In this chapter, analysis of the proposed system is presented. Comparative study of the proposed sMatch with previous work done by Q. Ren *et al.* [18] and Ahmad *et al.* [54] is presented in this chapter. The comparison is performed in terms of different aspects: run time complexity, hit ratio, and handling of incorrect and "SELECT *" type queries.

First of all, we compared the query matching complexity of proposed semantic indexing scheme with segment based semantic indexing scheme. We have computed the complexity of both the schemes as follows.

Theorem 5.1. *For a relation R having attribute set $A = \cup A_i$, where $1 \leq i \leq n$; query matching complexity for segment based scheme is $m \times n \times 2^n - 1$ where m is number of required attributes in user query and n is the number of total attributes in a relation.*

The proof for Theorem 5.1 will be constructive nature. We shall begin with a lemma.

Segment based query matching scheme depends on the number of segments and number of attributes in each segment. Number of segments in cache depends on the number of attributes in a relation. Possible segments for a relation having n attributes are proved in Lemma 5.2.

Lemma 5.2. *For a relation R having attribute set $A = \cup A_i$, where $1 \leq i \leq n$; the maximum number of segments is $2^n - 1$*

Proof. Number of attributes in a relation $R = n$ Number of subsets, P, for n attributes can be computed as given in Equation 5.1,

$$P(n) = 2n \tag{5.1}$$

Number of disjoint queries (D_{QU}) on relation will be equal to the subsets except the empty set.

$$D_{QU} = P(n) - \varnothing \tag{5.2}$$

There will only one empty subset in $P(n)$ namely $\varnothing$. By replacing values in Equation 5.2 from 5.1 we get;

$$D_{QU} = 2^n - 1 \tag{5.3}$$

As we know [18] that the number of segments $|S|$ on cache will be equal to the number of disjoint queries. i.e.

$$|S| = D_{QU} \tag{5.4}$$

By replacing the value into Equation 5.4 from 5.1, we get;

$$|S| = 2^n – 1 \tag{5.5}$$

Hence Lemma 5.2 is proved. □

Comparisons required to find a single attribute over segments $|S|$ in the worst case are proved in Lemma 5.3

Lemma 5.3. *For a relation R having attribute set $A = \cup A_i$, where $1 \leq i \leq n$; $n \times 2^n - 1$ number of comparisons (NoC_n) required to find a single attribute A_i over $|S|$*

Proof. Finding a single attribute A_i over $|S|$ is required to visit each and every attribute in each segment. For a single attribute in a relation; there can be only one segment (according to Lemma 5.2) and number of comparison (NoC_n) required for query matching will also one and can be computed as

$$NoC_n = 2^1 - 1 \tag{5.6}$$

For two attributes; there will be three segments (according to lemma 1.1).

Number of comparisons (NoCn), in this case, will be required for query matching are 4 and can be computed as

$$\begin{aligned} NoC_n &= 2^2 - 1 + (2^1 - 1) && (5.7)\\ &= 2^2 - 1 + 2^0(2^1 - 1) && (5.8)\\ &= 2^2 - 1 + 2^0(2^{2-1} - 1) && (5.9) \end{aligned}$$

For three attributes; there will be three segments and number of comparisons (NoC_n) required for query matching are 12 and can be computed as

$$\begin{aligned} NoC_n &= 2^3 - 1 + 2^2 - 1 + 2^1 - 1 + 2^1 - 1 && (5.10)\\ &= 2^3 - 1 + 2^2 - 1 + 2(2^1 - 1) && (5.11)\\ &= 2^3 - 1 + 2^0(2^2 - 1) + 2^1(2^1 - 1) && (5.12)\\ &= 2^3 - 1 + 2^0(2^{3-1-0} - 1) + 2^1(2^{3-1-1} - 1) && (5.13) \end{aligned}$$

For four attributes; there will be three segments and number of comparisons (NoCn) required for query matching are 32 and can be computed as

$$\begin{aligned} NoC_n &= 2^4 - 1 + 2^3 - 1 + 2^2 - 1 + 2^1 - 1 && (5.14)\\ &\quad +2^1 - 1 + 2^2 - 1 + 2^1 - 1 + 2^1 - 1 && (5.15)\\ &= 2^4 - 1 + 2^3 - 1 + 2^2 - 1 + 2^2 - 1 + 2^1 - 1 && (5.16)\\ &\quad +2^1 - 1 + 2^1 - 1 + 2^1 - 1 && (5.17)\\ &= 2^4 - 1 + 2^3 - 1 + 2(2^2 - 1) + 4(2^1 - 1) && (5.18)\\ &= 2^4 - 1 + 2^0(2^3 - 1) + 2^1(2^2 - 1) + 2^2(2^1 - 1) && (5.19)\\ &= 2^4 - 1 + 2^0(2^{4-1-0} - 1) + 2^1(2^{4-1-1} - 1) + 2^2(2^{4-2-1} - 1) && (5.20) \end{aligned}$$

Similarly, for five attributes; number of comparisons can be computed as:

$$\begin{aligned} NoC_n &= 2^5 - 1 + 2^0(2^4 - 1) && (5.21)\\ &\quad +2^1(2^3 - 1 + 2^2(2^2 - 1) + 2^3(2^1 - 1) && (5.22)\\ &= 2^5 - 1 + 2^0(2^{5-0-1} - 1) && (5.23)\\ &\quad +2^1(2^{5-1-1} - 1 + 2^2(2^{5-2-1} - 1) + 2^3(2^{5-3-1} - 1) && (5.24) \end{aligned}$$

For n attributes number of comparisons (NoC_n) can be computed as follows

$$\begin{aligned} NoC_n &= 2^n - 1 + 2^0(2^{n-0-1} - 1) + 2^1(2^{n-1-1} - 1) + 2^2(2^{n-2-1} - 1) + \quad (5.25) \\ & +2^{n-3}(2^{n-n-3-1} - 1) + 2^{n-2}(2^{n-n-2-1} - 1) \quad (5.26) \end{aligned}$$

We can write it as

$$\begin{aligned} NoC_n &= 2^n - 1 + \sum_{r=0}^{r=n-1} \left(2^r \left(2^{n-r-1} - 1\right)\right) && (5.27) \\ &= 2^n - 1 + \sum_{r=0}^{r=n-1} \left(2^r.2^{n-r-1}\right) - \sum_{r=0}^{r=n-1} 2^r = 2^n - 1 && (5.28) \\ & + \sum_{r=0}^{r=n-1} 2^{n-r+r-1} - \sum_{r=0}^{r=n-1} 2^r && (5.29) \\ &= 2^n - 1 + \sum_{r=0}^{r=n-1} 2^{n-1} - \sum_{r=0}^{r=n-1} 2^r && (5.30) \end{aligned}$$

By using geometric series we get

$$\begin{aligned} &= 2^n - 1 + 2^{n-1}. \sum_{r=0}^{r=n-1} 1 - \frac{2^n - 1}{n - 1} && (5.31) \\ &= 2^n - 1 + 2^{n-1}.n - 2^n + 1 && (5.32) \\ &= n.2^{n-1} && (5.33) \end{aligned}$$

Hence Lemma 5.3 proved. □

In Lemma 5.3, we have proved that query matching complexity of finding a single attribute over $|S|$ is $n \times 2^n - 1$

It is obvious that this complexity will be $m \times n \times 2^n - 1$ to find out the m attributes over relation R having n attributes. The above calculated comparisons are derived from the stored segments in cache. Now, if there are 'm' attributes required in user query, then total comparison will be $m \times n \times 2^n - 1$. Hence, the computation complexity of segments based will be $m \times n \times 2^n - 1$; where m is the number of attributes in user posed query and n is the number of attributes in a relation.

Hence Theorem 5.1 proved.

Corollary 5.4. *In the best case the query matching complexity for segment based scheme will be $m \times 2^n - 1$. In best case, each required attribute will be* *found out at first location of segment. According to Lemma 5.2, total numbers of segments* ($|S|$) *will be* $2^n - 1$*. For each attribute* m *we have to visit* $2^n - 1$ *segments. In this case query matching complexity will be* $m \times 2^n - 1$*.*

Theorem 5.5. *For a relation* R *having attribute set* $A = \cup A_i$*, where* $1 \leq i \leq n$*; query matching complexity for 6-HiSIS based scheme is* $m \times n$ *where* m *is the number of required attributes in user query and* n *is the number of total attributes in a relation.*

Proof. This proof is very straightforward; because 6-HiSIS indexes the semantic enabled schema instead of semantic segments. Number of attributes will always be n irrespective of the number of disjoint queries. The number of comparison (NoC_n) to find out m attributes over R will be $m \times n$ in worst case. □

As proved that query matching complexity is reduced from exponential to polynomial. In result of a faster query matching algorithm, data access latency for query is reduced.

Worst case complexity analysis of 6-HiSIS, graph based and segment based scheme given in Figure 5.1. We have assumed that each comparison will take one millisecond to compute.

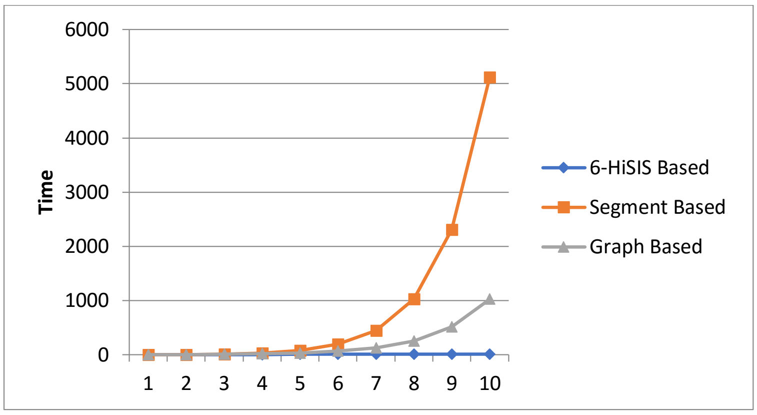

Figure 5.1: Worst case complexity analysis

Similarly; best case complexity analysis of 6-HiSIS and segment based scheme is given in Figure 5.2. We assumed that each comparison will take one millisecond to compute.

In Figure 5.3, comparison between segment based [18], graph based [54] and 6-HiSIS based schemes on the base of hit ratio is given. In this comparison, we assume that all of the data is available (hit ratio should be 100%. Now,

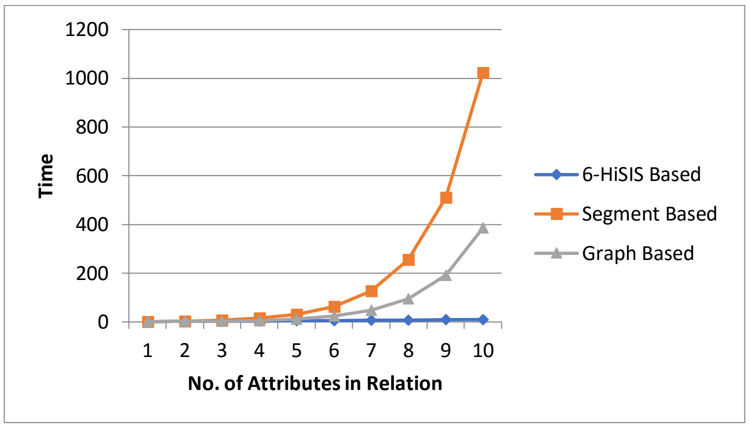

FIGURE 5.2: Best case complexity analysis

assume user poses queries with '*' operator (Select * type queries), which indicates that user needs all of the attributes. As soon as we will increase this type of queries hit ratio for graph based and segment based will decrease, while hit ratio for 6-HiSIS based scheme will remain at 100%. This happens because both Segment & Graph Based techniques are unable to handle "Select *" type queries and retrieve all the data from the remote server. Whereas, 6-HiSIS based scheme evaluates this type of queries by using schema. Therefore, hit ratio by 6-HiSIS remains at 100%.

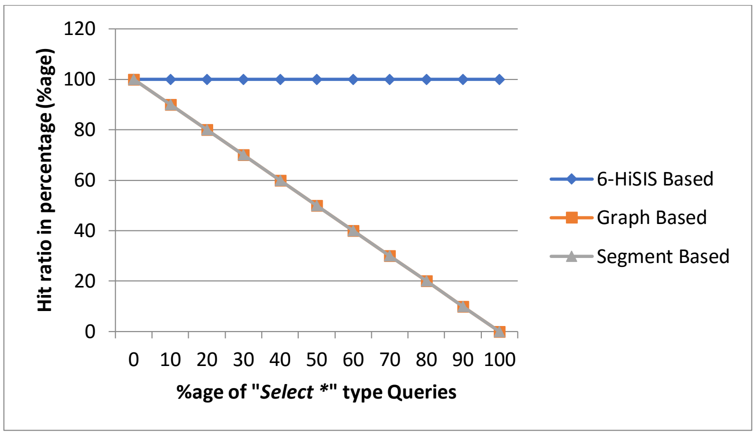

FIGURE 5.3: Hit Ratio Comparison by increasing Select * queries

In Figure 5.4, time comparison between segments based [18], graph based [54] and 6-HiSIS schemes on the basis of incorrect user queries is presented. Here, we assume that the user is going to pose incorrect queries. As soon as we will increase number of incorrect queries, computing time for graph based and segment based will be increased while hit ratio for 6-HiSIS based scheme will remain at 100%. This happens because both segment & graph based techniques are unable to handle the zero level rejection. Whereas, 6-HiSIS based scheme evaluates this type of queries by using schema.

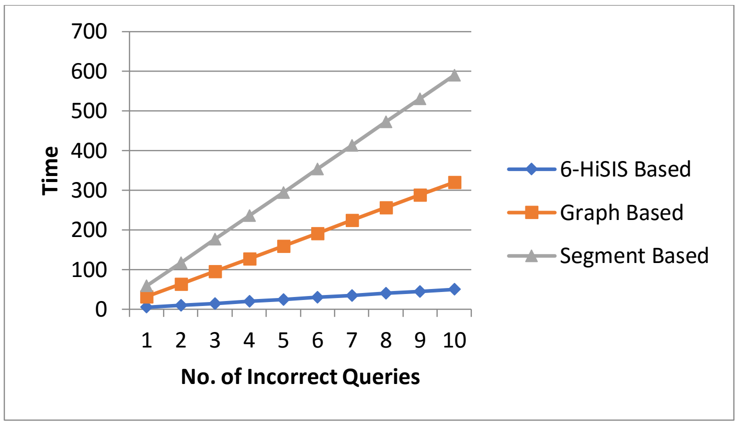

FIGURE 5.4: Time Comparison on the base of Incorrect Queries

In Figure 5.5, space complexities of segment [18], graph based [54] and 6-HiSIS based schemes [55] is compared. As we have discussed earlier in the proposed work section that space complexity will be higher than previous schemes until n attributes are retrieved or n distinct queries have been processed and their semantics are cached in semantic cache.

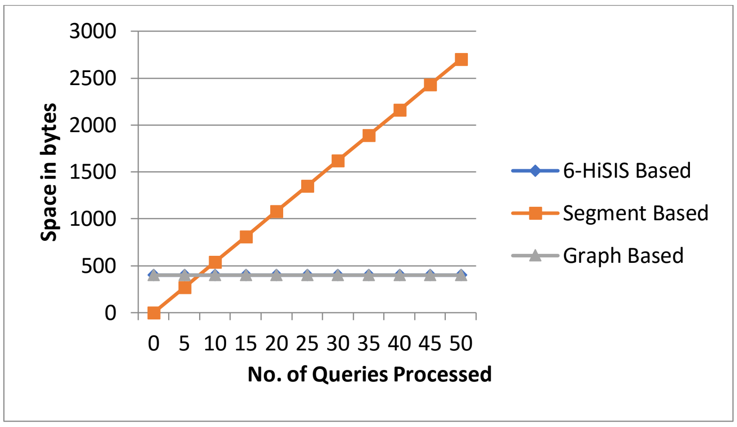

FIGURE 5.5: Space Complexity Analysis

Chapter 6

Case Study and Experimental Results

In this chapter, the validation of proposed technique is presented with the assistance of a case study and experimental results. Case study is utilized to explain and prove the correctness of the proposed algorithms in Chapter 4 and experimental results help to prove the efficiency and effectiveness of the proposed techniques. Question 6 from research questions (listed in Section 1.3) is answered in this chapter. This section is further divided into two subsections; Section 6.1 presents the validation of the proposed system with case study and 6.2 presents the experimental results.

6.1 Case Study

This section validates our proposed semantic indexing and query processing techniques with the help of a case study. Here we have used the schema of University instead of Library to ensure that proposed technique is generic rather than specific. Figure 6.1 presents the schema of a university with two relations employee and students having 4 and 3 fields respectively.

For the above given schema of university, there are 15 and 7 segments possible across employee and students relations respectively, according to previous work [18]. In simple words, we can say that there are 15 queries possible against employee and similarly 7 for students, as given in Table 6.1.

In the above example, there are 22 possible queries that make separate segments. So the formula to calculate possible segments across a single relation

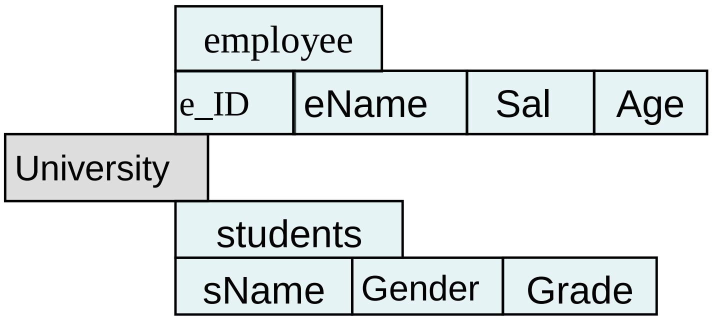

FIGURE 6.1: Schema for University Database

over n attributes is $2^n - 1$. Then add segments across each relation. As in example, $15 + 7 = 22 : (2^4 - 1 = 15 \& 2^3 - 1 = 7)$. Hence, 22 segments are to be visited to check the availability of data in the cache in the worst case which increases the response time drastically.

Schema based hierarchal scheme reduces the number of comparisons to find out whether data are available in the cache or not. Only n comparisons are required to check the availability of data in the cache. Table 6.2 can be rearranged according to our proposed schema based semantic indexing scheme as in Table 6.2.

Table 6.2 represents the structure of a schema based semantic indexing instead of actual contents. There is a need to compare/match only 4 and 3 fields instead of 15 and 7 segments respectively according to the previous schemes. Also it has the ability to reject invalid queries at the initial level instead of further processing.

For detailed discussion and simplicity, we consider only employee table of university database. Let us consider there is an employee table on a server with 4 fields defined in the university schema in Figure 6.1. An employee table on server is given in Table 6.3.

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

Case-I: Zero Level Rejection In this case we have taken an example that covers the query rejection at initial level.

SR	S	SA	SP	SC
Employees	S1	e_ID	Condition 1	1
	S2	eName	Condition 1	2
	S3	Sal	Condition 1	3
	S4	Age	Condition 1	4
	S5	e_ID, eName	Condition 3	5
	S6	e_ID, Sal	Condition 4	6
	S7	e_ID, Age	Condition 5	7
	S8	e_ID, eName Sal Age	Condition 6	8
	S9	e_ID Sal Age	Condition 7	9
	S10	e_ID, eName Age	Condition 8	10
	S11	eName Sal	Condition 9	11
	S12	eName Age	Condition 10	12
	S13	Sal Age, eName	Condition 11	13
	S14	e_ID, eName, Sal	Condition 12	14
	S15	Sal, Age	Condition 3	15
Students	S16	sName	Condition 21	16
	S17	Grade	Condition 21	17
	S18	Gender	Condition 21	18
	S19	Gender, Grade	Condition 22	19
	S20	Gender, sName	Condition 23	20
	S21	sName, Gender	Condition 24	21
	S22	sName, Grade, Gender	Condition 25	22

Table 6.1: Possible segments for given database

Database Name	Relation Names	Attribute Names	Attribute Status	Condition	Content Reference
University	Employee	eName	TRUE	Condition 1	1
		Age	FALSE	Null	Null
		Sal	TRUE	Condition 2	2
	Student	e_ID	TRUE	Condition 1	1
		Gender	FALSE	Null	Null
		Grade	FALSE	Null	Null
		sName	FALSE	Null	Null

Table 6.2: Schema Based Indexing

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	Mehreen	39	30450
117	Tabinda	39	28850
118	Yaseen	40	24450
119	Anees	45	30000
120	Komal	50	30000

TABLE 6.3: Employee Table in Database

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

TABLE 6.4: Contents on Cache in Case-I

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

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

Data on the cache will be in the form as given in Table 6.4.

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

1. *SELECT* eName, Age *FROM* employee
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 (project attribute is incorrect)

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

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

Table 6.5: Contents on Cache in Case-II

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, the list of projected attributes (eName, Age in first query), relation (employee in first query) and predicate attributes are checked from schema based indexing scheme; and query will be rejected due to unavailability of "employee" relation in the 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 queries.

The proposed semantic cache system never sends the incorrect queries to the server while previous semantic cache systems send the incorrect queries to the server, which results in increased network traffic and increased data access latency. So, due to zero level rejection, proposed system performs better to reduce the network traffic as well as to reduce the data access latency.

Case-II: SELECT * Handling In this case, we have taken 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 the cache.

$$\boxed{\textit{SELECT}\ \text{eName, Age}\ \textit{FROM}\ \text{employee}\ \textit{WHERE}\ \text{age} > 30}$$

Data for above query will be retrieved and stored on cache will be in the form as given in Table 6.5.

In the above query, e_ID is not required but retrieved. It is due to the requirement of a key-contained [18] contents. Now, let us assume that a user has posed the following query.

$$\boxed{\textit{SELECT}\ {}^*\ \textit{FROM}\ \text{employee}\ \textit{WHERE}\ \text{age} > 30}$$

Now, all of the fields of employee are required, but according to previous work common set will be calculated (intersection of cached attributes and user's query attributes). There is no way 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 that which of the attributes are not in the cache, and should be retrieved from the server.

Here again we need a schema at cache (first we need a schema for zero level query rejection). If a schema is available in the cache then SELECT CLAUSE with '*' can be handled easily.

By handling *SELECT ALL* type queries, utilization of available data will be maximized that result in improved hit ratio. Splitter splits the query and sends it to the rejecter. Rejecter checks the list of the fields with relation and predicate attribute from schema based indexing semantics. Query will not be rejected due to availability of all member of list in the schema. Common and difference set of attributes will be computed and shall be sent to the 1st level query matcher. i.e. C_A and D_A will be computed. Remainder query (rq_1) with difference attributes (here is only one difference attribute that is 'Sal') will be generated by 1st level query matcher like below.

rq_1 =*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 a 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 all, the semantics of predicate will be computed by semantic extractor as follow.

M_C = Age

N_{MCC} =NULL

N_{MCU} =NULL

D_{VU} =30

D_{VC} =30

Op_C =>

Op_U =>

After computation of predicate semantics, predicate for probe and remainder query will be computed by using an ExplicitSemanticMatching algorithm (actually 112 rules are used here). At first, main class of algorithm is selected, here data value of user ($D_{VU} = 30$) is equal to the data value of cache ($D_{VC} = 30$). So, the algorithm will be executed as given in Algorithm 11.

Algorithm 11 Algorithm executed for Case II

```
1: if $D_{VC}[i] = D_{VU}[i]$ then
2:     if $(((O_{PC}[i] \in \{\geq\}) \wedge (O_{PU}[i] \in \{>, =\})) \vee (O_{PC}[i] = O_{PU}[i]) \vee ((O_{PC}[i] \in \{\leq\}) \wedge (O_{PU}$
   then
3:         $C_![i] \leftarrow C_C O_{PC} D_{VC}$ $NC_1[i] \leftarrow$NULL;
4:     end if
5: end if=0
```

Predicate for probe and remainder will be computed we say it C_1 (for cached) and N_{C1} (for non-cached).

$$C_1 = \text{Age} > 30$$
$$N_{C1} = \text{Null}$$

Due to simple predicate, rules defined for complex queries will not be applied. Finally, ImplicitSemanticMatching 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 ImplicitSemanticMatching algorithm will be applied.

If $(N_{MC} = \text{Null})$ and $(N_{MU} = \text{Null})$ **then**

$$C_2 = C_1$$
$$N_{C2} = N_{C1}$$

There will be no change in the predicate of probe and remainder query. Then, probe query (pq) and second remainder query (rq_2) will be generated as below.

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

rq_2 =Null

In the last step results of rq_1, pq and rq_2 are combined by rebuilder.

Due to handling of *SELECT ALL* type queries, the proposed system maximizes the utilization of available data at cache, while the existing systems are not able to handle these types of queries, which result in minimum utilization of available data at cache. Maximum utilization of available data on cache results in increased data availability, decreased network traffic (as minimum data will be retrieved from the server), decreased data access latency.

Case-III: Generation of Amending Query $(Q_U \subseteq Q_C)$

In this case the generation of amending query is elaborated.

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

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

TABLE 6.6: Contents on Cache in Case-III

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

Data for above query will be retrieved and stored in the cache will be as given in Table 6.6.

Note that e_ID is not required, but retrieved. It is due to the requirement of a key-contained [18] contents. Now, let us assume that a 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 the case-II.

Note that data across eName and Sal is available in the cache, but predicate attribute (Age) is not available. Now, one cannot select data from cache due to absence of the predicate attribute, because one cannot decide that which of the data satisfies the selection criteria (Age> 35). To solve this problem, another query called amending query [18] to retrieve primary attribute from server on user selected criteria (as below) will be generated.

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

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

Case-III: Generation of Amending Query (Exact Match)

In this case, generation of amending query is elaborated in which user query is exactly equivalent to the cached query.

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

e_ID	**eName**	**Sal**
115	Komal	17000
116	Mehreen	30450
117	Tabinda	28850
118	Yaseen	24450
119	Anees	30000
120	Komal	30000

TABLE 6.7: Contents on Cache in Case-IV

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

Data for above query will be retrieved and stored in the cache will be as given in Table 6.7.

Note that e_ID is not required, but retrieved. It is due to the requirement of key-contained [18] contents. Now, let us assume that a user has posed the following query which is exactly similar to the cached query.

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

Here, only the generation of amending query has been discussed. Remaining procedure will be same as discussed in Case-II.

Algorithm to generate amending query proposed by Ren *et al.* perform same processing as done in Case-III and generates amending query as follow.

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

In fact, there is no need of amending query in this scenario because following query can be executed on cache without amending query with accurate result.

pq =*SELECT* eName, Sal *FROM* cache-ref

As we know that in cache only those rows are stored whose age¿35 and in user query all those rows are required whose age¿35. So this query will generate the accurate result from cache without retrieving anything from the server. Due to this network traffic and data access latency will be reduced.

Case-V: Generation of Amending Query $(Q_C \subseteq Q_C)$

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 the cache.

e_ID	eName	Sal
116	Mehreen	30450
117	Tabinda	28850
118	Yaseen	24450
119	Anees	30000
120	Komal	30000

TABLE 6.8: Contents on Cache in Case-V

SELECT eName *FROM* employee *WHERE* age> 37

Data for above query will be retrieved and stored in the cache will be as given in Table 6.8.

Note that e_ID is not required, but retrieved. It is due to the requirement of key-contained [18] contents. Now let us assume that a user has posed the following query.

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

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

Algorithm to generate amending query proposed by Ren *et al.* [18] perform same processing as done in Case-III and generates amending query as follows.

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

In fact, there is no need of amending query in this scenario because following probe query can be executed on cache without amending query with a remainder query to retrieve data which is not available in the cache.

pq =*SELECT* eName *FROM* cache-ref
rq =*SELECT* eName,Sal *FROM* employee *WHERE* Age> 35 and Age< 37

As we know that in cache only those rows are stored whose age¿37 and in user query all those rows are required whose age¿35. So this query will generate the accurate result from cache with result of remainder query. As algorithm by Ren et al. generates amending query in this case, while our proposed algorithm solves this query without any amending query. Due to this network traffic and data access latency will be reduced as compared to the semantic cache system by Ren *et al.* [18].

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

TABLE 6.9: Contents on Cache in Case-VI

Case-VI: Hidden Semantics

Here efficient predicate matching to improve the hit ratio by using Implicit-SemanticMatching algorithm is elaborated with an example.

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

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

Data for above query will be retrieved and stored in the cache as given in Table 6.9.

Note that e_ID is not required, but retrieved. It is due to the requirement of a key-contained [18] contents. Now let us assume that a user has posed the following query.

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

Here all of the required fields are matched with cached query. Splitter splits the query and sends it to the rejecter. Rejecter checks the list of the fields with relation and predicate attribute from schema based indexing semantics. Query will not be rejected due to availability of all member of list in the 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 (rq_1) will be null due to an empty set of difference attributes (All required attributes exist in the cache). So, remainder query by 1^{st} level query matcher will be like below.

$$rq_1 = Null$$

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

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

M_C = NULL
N_{MC} =Age
N_{MU} =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 is based on these rules:

C_1 = Null
N_{C1} =Null

Due to simple predicate, rules defined for complex queries will not be applied. Finally, ImplicitSemanticMatching algorithm will be applied to generate final predicate for probe and remainder queries. As it is computed, N_{MC} and N_{MC} both are not null. So, fourth case of ImplicitSemanticMatching algorithm will be applied.

4. **Else If**(N_{MC} ≠null) and (N_{MU} ≠null) **then** a.
C_1=(C_1) + (N_{MU}) + (N_{MC}) b.
N_{C2}=(C_1) + R (N_{MC}) + (N_{MU}) ∨ ((N_{C1}) + (N_{MU}))

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

C_1:=(Age> 30 and (eName='Komal')
C_1:=(Age≤ 30 and (eName='Komal')

Then, probe query (pq) and second remainder query (rq_2) will be generated as below.

pq =*SELECT* eName, Age *FROM* employee *WHERE* Age> 30 and (eName='Komal')
rq_2 =*SELECT* eName, Age *FROM* employee *WHERE* Age≤ 30 and (eName='Komal')

In the last step, results of rq_1, pq and rq_2 are combined by rebuilder.

Due to having the ability to find hidden semantics, proposed system utilizes the maximum data available data in the cache, which results in minimum data retrieval from the server. As a proposed system retrieves lesser data from the server as compared to the existing system, therefore, the proposed systems perform better to reduce the data access latency, reduce the network traffic and increases the availability of the system.

6.2 Experimental Results

In this section we have examined the performance of proposed system. In order to do this, we created a database with a relation Product having 22 attributes; id, category, itemname, barcode, manufacturingPrice, shipingcost, salestax, totalprice, purchaseprice , salesprice, discount, profit, itemdes, vendor, itemweight, itemsize, quantity, location, expdate, manfdate, ingredients, and manufacturer.

We have inserted 10 million records into product. Total size of database on memory is 14 GB.

We have issued 10-12 different queries in each scenario. Each query is issued 10 times and noted the latency time, each time then averaged the result. Size for each query is varied from scenario to scenario.

Memory allocated to Semantic cache is 4GB.

Experimental activity has been done in two directions; in first complete database and semantic cache has been managed at same machine and in second database was located on one machine and semantic cache was managed on second machine.

We have implemented three systems;

- without semantic cache
- Proposed by Ren et al. (existing system)
- Proposed system (sCacheQP).

Without semantic cache mean we have executed queries without managing any caching. In case of existing system, we have implemented technique proposed by Q. Ren et al (we have used Q. Ren in legends of graphs in next sections) which is the only complete available system for query processing over semantic cache.

In the following sub sections we have presented the scenarios and results. Results in graph are based on 10-12 discrete values. However, these discrete values are joined to show the trend.

6.2.1 Database and Semantic Cache at Same Machine

In this section, we present the efficeincy of the proposed system by developing a prototype. In the developed enviornment, we have maintained original database and cache at same machine, and then examined the performance of system (proposed, existing and without semantic cache).

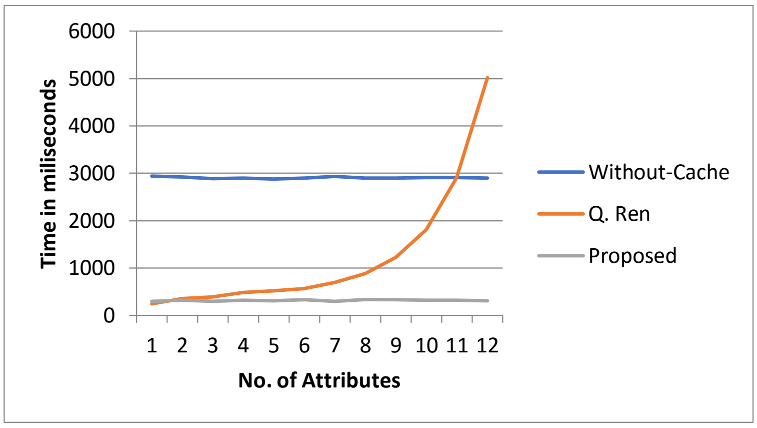

Figure 6.2: Analysis with increasing number of segments

In Figure 6.2, we have presented the analysis by fixing the hit ratio as 100% and growing the segments form 1 to 4095. On x-axis, number of attributes are increasing. As we discussed and proved in Chapter 5, that $2^n - 1$ (here n is number of attributes) segments can be created. With increase in number of attributes, number of segments increased exponentially [55]. System proposed by Ren et al. [18] perform matching with each of the segments in semantic cache which results in increased data access latency with increasing number of segments on cache. Our proposed semantic cache system reduces the data access latency as depicted in Figure 6.2.

In the next scenario, techniques are compared with respect to growth of required data by user. In this scenario, we fixed the number of segments which are 100 and increased the size of query in term of required data. Size of query is varying from 100000 records to 1000000 records. Result is presented in Figure 6.3.

In the following scenario, we compared the techniques by fixing the no. of segments and do variation in hit ratio. Hit ratio varies from 10 percent to 100 percent. Result is depicted in Figure 6.4. In result we have examined that semantic cache performs worse with a decrease in hit ratio.

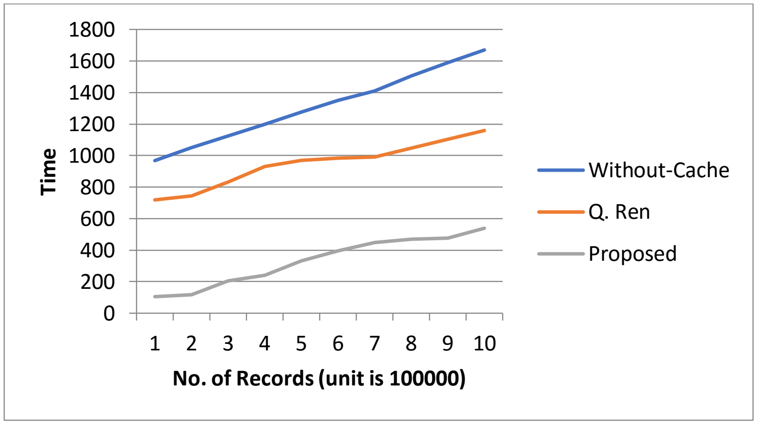

FIGURE 6.3: Analysis with increasing number of records

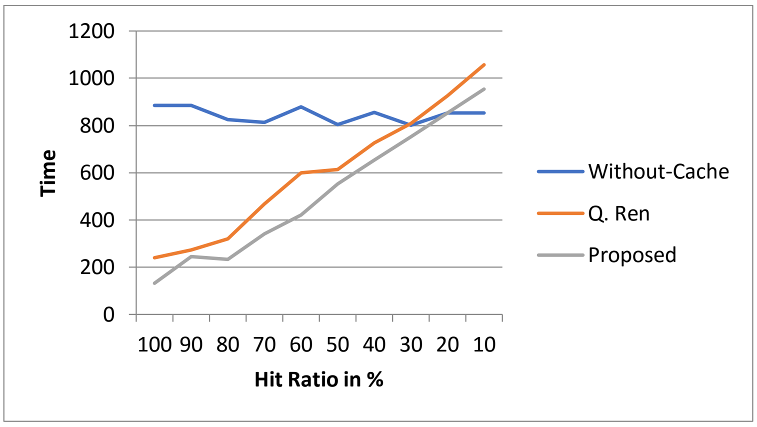

FIGURE 6.4: Analysis for Varying Hit Ratio

Advantage of amending query is depicted in Figure 6.5. In this scenario we have established a situation where user queries are exactly similar to cached queries, but data about predicate attributes is not stored in cached queries. One of the examples for cached and user query is given below:

Cached query: Select profit, discount from product where salprice> 200

and

User query: Select profit, discount from product where salprice> 200

In this case, data is stored for profit and discount not for salprice. In this situation existing system [18] generates amending query in which data for salprice is retrieved from original database. In fact, there is no need of amending query (data about salprice) in this case. Proposed system did not generate

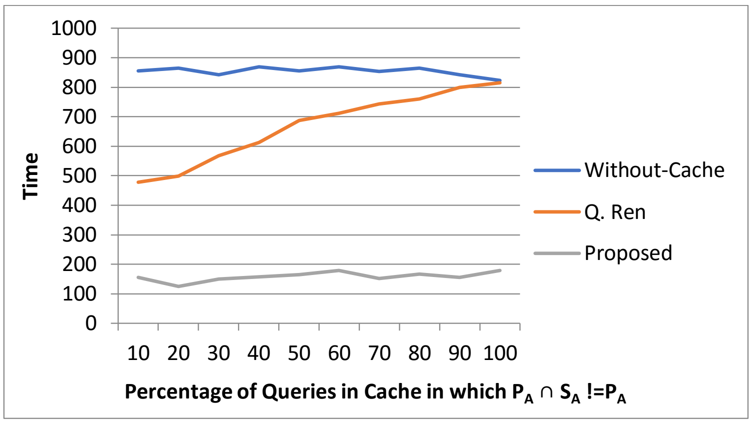

FIGURE 6.5: Analysis for Exact Match

any amending query due to exact match. Due to that the proposed technique performs better as given in Figure 6.5.

In next scenario, we examined the performance of the system for sub set match. In other words in this case user's posed queries are sub set of cached query. In this case, existing system generates amending query but our proposed system have no need of amending query. Results are given in Figure 6.6.

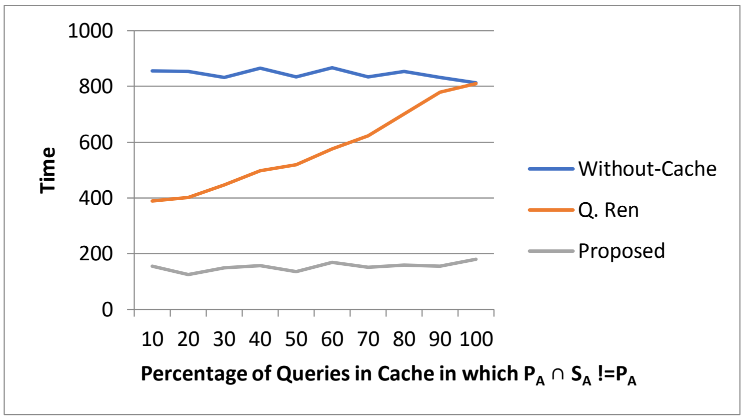

FIGURE 6.6: Analysis for Subset Match

In the above two cases, hit ratio is 100 percent. In a scenario, where conditioned attribute is absent, and hit ratio is not 100 percent, as shown in Figure 6.7. In this case, both techniques (proposed and Q. Ren) generate amending queries. Results are presented in Figure 6.7

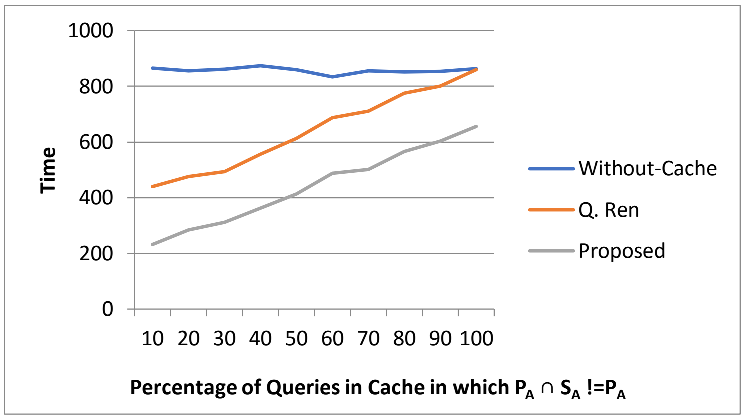

FIGURE 6.7: Analysis for Partial Match

6.2.2 Database and Semantic Cache at Different Machine

In this case, we stored database at one machine and managed cache at another machine and analyzed the performance of the system.

In Figure 6.8, we presented the analysis by fixing the hit ratio as 100% and growing the segments form 1 to 4095. System proposed by Ren *et al.* manages semantics in segments [18] and extract informatin by matching the semtnics of user query with semantics stored in segments. As the number of segments grows exponentially [55] which results in exponential query matching. Due to the exponential query matching process, data access time increased exponentially with increase in segments exponentially as depicting in Figure 6.8. Due to efficient semantic indexing scheme by our proposed system, query matching performed in polynomial time [55]. Due to faster query matching process, we achieved reduced data access latency as shown in Figure 6.8.

In the next scenario, techniques are compared with respect to growth of required data by the user query. Here, we have fixed the number of segments which are 100 and increased the size of query in term of required data. Size of the query is varying from 100000 records to 1000000 records. The result is presented in Figure 6.9.

Next, we compared the techniques by fixing the number of segments and did variation in the hit ratio. Hit ratio varies from 10 percent to 100 percent. The result is depicted in Figure 6.10. The results show that the semantic cache performs nearly equal to the system without cache with decrease in hit ratio, ie. 10% hit ratio.

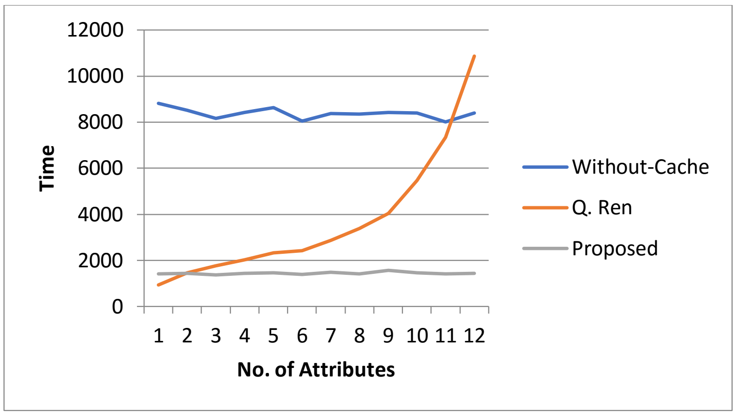

FIGURE 6.8: Analysis with increasing number of segments

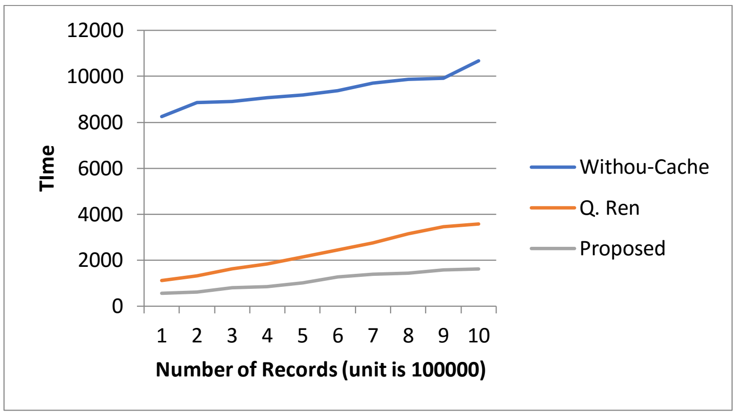

FIGURE 6.9: Analysis with increasing number of records

Advantage of amending query is depicted in Figure 6.11. In this scenario, we have designed queries, which are exactly similar to stored query in cache but these queries have not stored conditioned attribute. In case of existing system, amending query will be generated, but proposed system will not generate amending query due to the exact match. Therefore, the proposed technique performs better, as given in Figure 6.11.

In Figure 6.12, the results of the system for sub set match are shown. Here, the users posed queries are sub set of cached queries. The existing system generates amending query, but our proposed system does not require amending query.

In above two cases, hit ratio is 100 percent. In a scenario where conditioned attribute is absent as well as hit ratio is not 100 percent, as shown in Figure 6.13. In this case, both techniques (proposed and Q. Ren) generates amending queries.

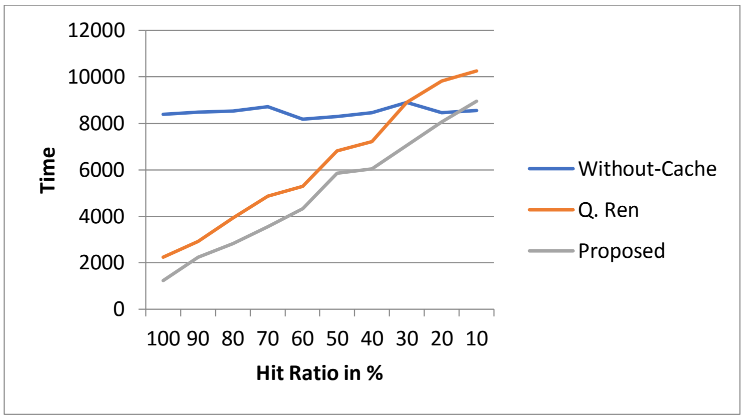

FIGURE 6.10: Analysis for Varying Hit Ratio

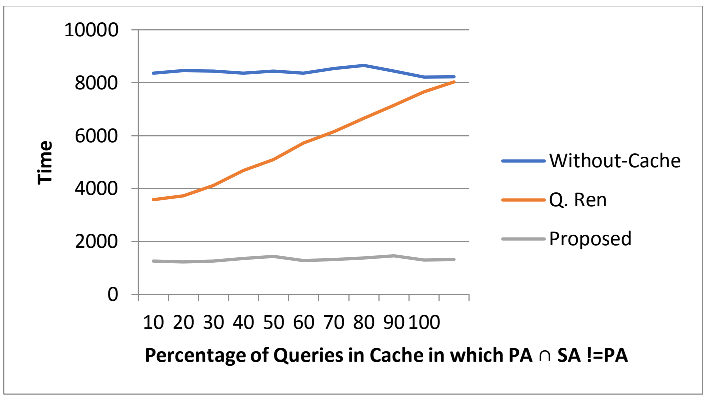

FIGURE 6.11: Analysis for exact match

Implemented systems achieve the all four points (Accuracy, increased data availability, reduced network traffic, and reduced data access latency) of benchmark. Let us discuss how each point is achieved by implemented system.

Accuracy: Proposed implemented system ensures accuracy by generating accurate result, nothing more or less than the user required result. Implemented system generates probe and remainder queries accurately. Accuracy of the system is verified by experts.

Increased Data Availability: Data Availability is increased by reducing the redundant semantics and by maximizing the utilization of available data in the cache. Our system generates probe and remainder query in such a way, that data is never retrieved from the server if this available in the cache. This is verified by experts

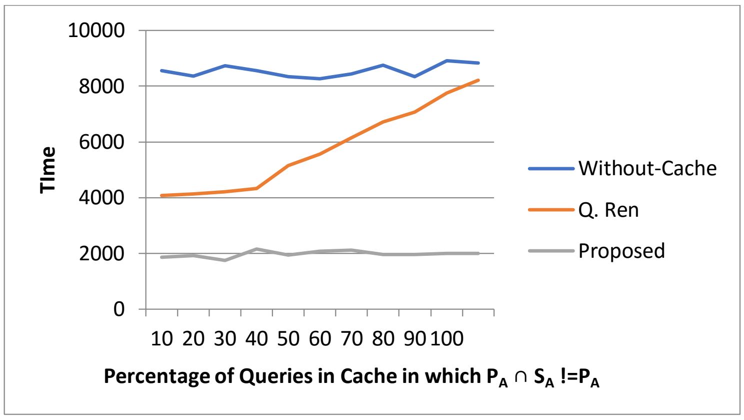

FIGURE 6.12: Analysis for Subset Match

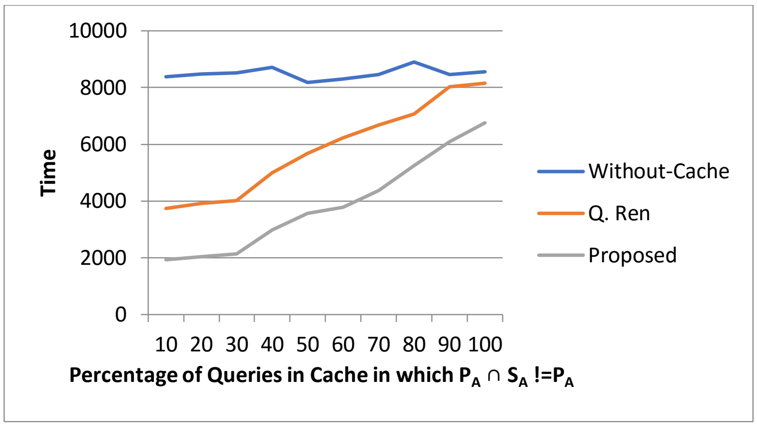

FIGURE 6.13: Analysis for Partial Match

Reduced Network Traffic and Reduced Data Access Latency: Through implementation, it is verified by results (from Figure 6.2 to 6.3) that proposed system reduces the data access latency. This reduced data access latency becomes possible due to maximizing the utilization of available data from cache. Due to maximum utilization of available data, there was least data retrieved from the server, which results in reduced network traffic.

Chapter 7

Conclusion and Future Work

Cache is an important service to reduce the data access latency, especially for the retrieval of same data frequently in a distributed environment. Reduction in data access latency is directly proportional to the increase in hit ratio. Semantic cache has a potential to maximize the hit ratio. There are multiple components of a semantic cache system, such as, indexing of semantics of stored queries, query processing, and contents storage. In this study, a framework to evaluate the semantic cache system and the impact of different parameters on the performance, have been presented. Serious limitations of the existing approaches have been identified.

Assessment of Research Questions

Let us first conclude the answers of the research questions presented in Chapter 1. Firstly, what should be the criterion to evaluate the existing semantic cache systems? After critically analyzing the literature, we could not find any comprehensive criteria to evaluate a semantic cache system. In order to maximize the hit ratio, a cache system must ensure four relations given in Equation Equation 7.1, 7.2, 7.3 and 7.4:

$$\text{Data}_{\text{ReminderQuery}} \cup \text{Data}_{\text{ProbeQuery}} \implies \text{Data}_{\text{UserQuery}} \tag{7.1}$$

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{Cache}} \implies \varnothing \tag{7.2}$$

$$\text{Segment}_{i\text{-cache}} \cap \text{Segment}_{j\text{-cache}} \implies \varnothing \forall i, j \ \text{ whenever } i \neq j \tag{7.3}$$

$$\text{Data}_{\text{ReminderQuery}} \cap \text{Data}_{\text{ProbeQuery}} \implies \varnothing \tag{7.4}$$

Second question was how existing system should be evaluated on the defined criterion? In order to ensure the above four expressions (rules), we have extracted a few parameters from a cache system and their relation with the above defined criterion; the parameters are SELECT ALL handling, zero level rejection, predicate handling, generation of amending query, query matching complexity, indexing structure and query processing complexity that influence the efficiency of a query processing algorithm.

Thirdly, what are the limitations in the existing systems? The in-depth analysis of literature revealed that there are some serious limitations in terms of these parameters. Due to their indexing approach for storing the semantics, the complexity of query matching may grow exponentially. Query trimming has limitations due to the use of the satisfiability & implication algorithm. Zero level rejection and generation of amending query with scheme enabled semantic storage are not available in these schemes.

The next question was how a semantic cache system should be designed to overcome the limitations in existing systems? A semantic caching architecture has been proposed to process the SELECT and PROJECT queries over relational distributed databases.

Question five was: what are the algorithms (with reduced complexity) to overcome the limitations of existing systems? We have formulated algorithms for sCacheQP; Splitter Algorithm, Rejecter Algorithm, 1^{st} Level Query Rewriter, Semantic Extractor, ExplicitSemanticMatching, ImplicitSemanticExtractor, 2^{nd} Level Query Rewriter, Generation of Amending Query, and Predicate Merging.

Finally, in which area the proposed system performs better than existing systems? The proposed technique has been implemented and results are scrutinized carefully. On the basis of results, it is concluded that the proposed technique performs better in term of producing correct results, reducing data access time, increasing data availability, and reducing network traffic for the queries with overlapped results (common data in multiple queries).

Accuracy: Accuracy is achieved by designing, accurate predicate matching algorithm efficient semantic indexing scheme. The efficient semantic indexing scheme ensures to store complete and accurate semantics for already executed queries in the cache. Accurate and complete stored semantics results in accurate semantic cache system as discussed in Chapter 2. In Chapter 6, results of implemented system also depict that proposed system is more accurate than the existing systems due to accurate predicate matching. Accurate semantic cache system results in reducing the data access latency, as depicted in the results in Chapter 6.

Increased Data Availability: Data availability of proposed system is increased by developing an effacing semantic indexing scheme, by developing efficient algorithm to generate amending query, and by providing the facility to *SELECT ALL* handling. Increased data availability ultimately reduces the data access latency, which is depicted in results presented in Chapter 6.

Reduced Network Traffic: Network traffic is reduced in proposed system, by developing efficient query matching process, which has ability to reject incorrect queries at initial level, by developing an efficient algorithm to generate amending query, by handling *SELECT* '*' type queries, and due to design of efficient semantic indexing scheme. Due to reduction in network traffic, data access latency is reduced, which can be witnessed through the results in Chapter 6.

Reduced Data Access Latency: Data access time for the semantic cache system is reduced by efficient semantic indexing structure reduces the complexity of query matching process significantly as the incoming query will have to be matched with stored semantics. Faster predicate matching ensures quick response time and reduced data access latency. Proposed semantic cache system facilitates zero level rejection, due to which, the response time is reduced as the incorrect queries are rejected at the local machine. Amending query and *SELECT ALL* handling maximizes the utilization of cached data, which results in reduction in data access latency. Implemented system depicts that proposed system performs better to reduce the data access latency as compared to the existing systems, as given in Chapter 6.

For future direction, semantic caching management and data replacement policies is an important area to explore. By current region-based data replacement policies; cache often results in poor cache utilization, because it discards entire regions from the cache. We are currently studying the use of region "shrinking" as a technique to alleviate this problem. In this study, we have examined the utility of orthodox value, as well as some semantic value in old-fashioned workloads and a mobile navigation workload. Our future plan comprises of further development of semantic value functions for this and other applications. An interval based predicate matching algorithm is another option which needs to be explored.

Bibliography

[1] A. A. Amer and A. A. Sewisy, "An extended technique for data partitioning and distribution in distributed database systems (ddbss)," Journal of Communications Technology, Electronics and Computer Science, vol. 12, pp. 13–19, 2017.

[2] N. H. Ryeng, J. O. Hauglid, and K. Nørvåg, "Site-autonomous distributed semantic caching," in Proceedings of the 2011 ACM Symposium on Applied Computing. ACM, 2011, pp. 1015–1021.

[3] C. Coronel and S. Morris, Database systems: design, implementation, & management. Cengage Learning, 2016.

[4] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "Pocketweb: instant web browsing for mobile devices," in ACM SIGARCH Computer Architecture News, vol. 40, no. 1. ACM, 2012, pp. 1–12.

[5] D. A. Patterson and J. L. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan kaufmann, 2017.

[6] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pp. 1920–1948, 2015.

[7] R. E. Hooker, D. R. Reed, J. M. Greer, C. Eddy, and A. J. Loper, "Fully associative cache memory budgeted by memory access type," May 16 2017, uS Patent 9,652,400.

[8] S. B. Edlund, M. L. Emens, R. Kraft, and P. C.-S. Yim, "Labeling and describing search queries for reuse," Nov. 19 2002, uS Patent 6,484,162.

[9] M. Qiu, Z. Ming, J. Li, K. Gai, and Z. Zong, "Phase-change memory optimization for green cloud with genetic algorithm," IEEE Transactions on Computers, vol. 64, no. 12, pp. 3528–3540, 2015.

[10] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan et al., "Semantic data caching and replacement," in VLDB, vol. 96, 1996, pp. 330–341.

[11] Q. Fan, K. Zeitouni, N. Xiong, Q. Wu, S. Camtepe, and Y.-C. Tian, "Nash equilibrium-based semantic cache in mobile sensor grid database systems," IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2016.

[12] Y. Lu and W. Wang, "Abortable transactions using versioned tuple cache," Jun. 26 2015, uS Patent App. 14/752,175.

[13] M. F. Bashir, R. A. Zaheer, Z. M. Shams, and M. A. Qadir, "Scam: Semantic caching architecture for efficient content matching over data grid," in Advances in Intelligent Web Mastering. Springer, 2007, pp. 41–46.

[14] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems," ACM Transactions on Database Systems (TODS), vol. 9, no. 4, pp. 503–525, 1984.

[15] M. U. Ahmed, R. A. Zaheer, and M. A. Qadir, "Intelligent cache management for data grid," in Proceedings of the 2005 Australasian workshop on Grid computing and e-research-Volume 44. Australian Computer Society, Inc., 2005, pp. 5–12.

[16] P. Godfrey and J. Gryz, "Semantic query caching for hetereogeneous databases." in KRDB, 1997, pp. 6–1.

[17] M. Ahmad, M. A. Qadir, and M. Sanaullah, "Query processing over relational databases with semantic cache: A survey," in Multitopic Conference, 2008. INMIC 2008. IEEE International. IEEE, 2008, pp. 558–564.

[18] Q. Ren, M. H. Dunham, and V. Kumar, "Semantic caching and query processing," IEEE transactions on knowledge and data engineering, vol. 15, no. 1, pp. 192–210, 2003.

[19] S. Ghandeharizadeh, J. Yap, and H. Nguyen, “Strong consistency in cache augmented sql systems,” in Proceedings of the 15th International Middleware Conference. ACM, 2014, pp. 181–192.

[20] T. Lahiri, M.-A. Neimat, and S. Folkman, “Oracle timesten: An in-memory database for enterprise applications.” IEEE Data Eng. Bull., vol. 36, no. 2, pp. 6–13, 2013.

[21] P.-A. Larson, J. Goldstein, and J. Zhou, “Mtcache: Transparent mid-tier database caching in sql server,” in Data Engineering, 2004. Proceedings. 20th International Conference on. IEEE, 2004, pp. 177–188.

[22] M. A. Alghobiri, H. U. Khan, T. A. Malik, and S. Iqbal, “A comprehensive framework for the semantic cache systems,” International Journal of Advanced and Applied Sciences, vol. 3, no. 10, pp. 72–78, 2016.

[23] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden et al., “The design and implementation of modern column-oriented database systems,” Foundations and Trends® in Databases, vol. 5, no. 3, pp. 197–280, 2013.

[24] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, “Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study,” ACM SIGMOD Record, vol. 39, no. 3, pp. 5–10, 2011.

[25] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, “Benchmarking oltp/web databases in the cloud: The oltp-bench framework,” in Proceedings of the fourth international workshop on Cloud data management. ACM, 2012, pp. 17–20.

[26] M. Sumalatha, V. Vaidehi, A. Kannan, M. Rajasekar, and M. Karthigaiselvan, “Hash mapping strategy for improving retrieval effectiveness in semantic cache system,” in Signal Processing, Communications and Networking, 2007. ICSCN’07. International Conference on. IEEE, 2007, pp. 233–237.

[27] ——, “Dynamic rule set mapping strategy for the design of effective semantic cache,” in Advanced Communication Technology, The 9th International Conference on, vol. 3. IEEE, 2007, pp. 1952–1957.

[28] M. F. Bashir and M. A. Qadir, “Hisis: 4-level hierarchical semantic indexing for efficient content matchingover semantic cache,” in Multitopic Conference, 2006. INMIC’06. IEEE. IEEE, 2006, pp. 211–214.

[29] X.-H. Sun, N. N. Kamel, and L. M. Ni, “Processing implication on queries,” IEEE Transactions on Software Engineering, vol. 15, no. 10, p. 1168, 1989.

[30] S. Guo, W. Sun, and M. A. Weiss, “Solving satisfiability and implication problems in database systems,” ACM Transactions on Database Systems (TODS), vol. 21, no. 2, pp. 270–293, 1996.

[31] M. Ahmad, M. A. Qadir, M. Sanaullah, and M. F. Bashir, “An efficient query matching algorithm for relational data semantic cache,” in Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on. IEEE, 2009, pp. 1–6.

[32] P. M. Kumar, T. Das, and D. Vaideeswaran, “Survey on semantic caching and query processing in databases,” in Proc. of the Second Intl. Conf. on Advances in Computer, Electronics and Electrical Engineering–CEEE 2013, 2013.

[33] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li, “Active query caching for database web servers,” WebDB (Selected Papers), vol. 1997, pp. 92–104, 2000.

[34] C. Lubbe, A. Brodt, N. Cipriani, M. Großmann, and B. Mitschang, “Disco: A distributed semantic cache overlay for location-based services,” in Mobile Data Management (MDM), 2011 12th IEEE International Conference on, vol. 1. IEEE, 2011, pp. 17–26.

[35] S. Cluet, O. Kapitskaia, and D. Srivastava, “Using ldap directory caches,” in Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 1999, pp. 273–284.

[36] A. Peters and A. Heuer, “Blues: usability of semantic caching approaches in pervasive ad-hoc scenarios,” in Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments. ACM, 2011, p. 33.

[37] E. Benson, A. Marcus, D. Karger, and S. Madden, “Sync kit: a persistent client-side database caching toolkit for data intensive websites,” in Proceedings of the 19th international conference on World wide web. ACM, 2010, pp. 121–130.

[38] J. R. Thomsen, M. L. Yiu, and C. S. Jensen, “Effective caching of shortest paths for location-based services,” in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012, pp. 313–324.

[39] B. B. Cambazoglu, I. S. Altingovde, R. Ozcan, and Ö. Ulusoy, “Cache-based query processing for search engines,” ACM Transactions on the Web (TWEB), vol. 6, no. 4, p. 14, 2012.

[40] G. Soundararajan and C. Amza, “Using semantic information to improve transparent query caching for dynamic content web sites,” in Data Engineering Issues in E-Commerce, 2005. Proceedings. International Workshop on. IEEE, 2005, pp. 132–138.

[41] M. Sumalatha, V. Vaidehi, A. Kannen, M. Rajasekar, and M. Karthigaiselven, “Xml query processing–semantic cache system,” IJCSNS, vol. 7, no. 4, p. 164, 2007.

[42] L. Chen, E. A. Rundensteiner, and S. Wang, “Xcache: a semantic caching system for xml queries,” in Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, 2002, pp. 618–618.

[43] M. Sanaullah, M. A. Qadir, and M. Ahmad, “Scad-xml: Semantic cache architecture for xml data files using xpath with cases and rules,” in Multitopic Conference, 2008. INMIC 2008. IEEE International. IEEE, 2008, pp. 361–367.

[44] C. Liu, B. C. Fruin, and H. Samet, “Sac: Semantic adaptive caching for spatial mobile applications,” in Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. ACM, 2013, pp. 174–183.

[45] C. M. Chen and N. Roussopoulos, "The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching," in International Conference on Extending Database Technology. Springer, 1994, pp. 323–336.

[46] H. Wan, X.-W. Hao, T. Zhang, and L. Li, "Semantic caching services for data grids," Lecture notes in computer science, pp. 959–962, 2004.

[47] J. Cai, Y. Jia, S. Yang, and P. Zou, "A method of aggregate query matching in semantic cache for massive database applications," in International Workshop on Advanced Parallel Processing Technologies. Springer, 2005, pp. 435–442.

[48] B. T. Jonsson, M. Arinbjarnar, B. Thorsson, M. J. Franklin, and D. Srivastava, "Performance and overhead of semantic cache management," ACM Transactions on Internet Technology (TOIT), vol. 6, no. 3, pp. 302–331, 2006.

[49] M. Bashir and M. Qadir, "Proq–query processing over semantic cache for data grid," Center for Distributed and Semantic Computing, 2007.

[50] C. Ehlers and B. Freitag, "Top-k semantic caching." Ph.D. dissertation, University of Passau, 2015.

[51] M. Xie, L. V. Lakshmanan, and P. T. Wood, "Efficient top-k query answering using cached views," in Proceedings of the 16th International Conference on Extending Database Technology. ACM, 2013, pp. 489–500.

[52] M. Ahmad, M. A. Qadir, T. Ali, M. A. Abbas, and M. T. Afzal, "Semantic cache system," in Semantics in Action-Applications and Scenarios. InTech, 2012.

[53] S.-W. Kang, J. Kim, S. Im, H. Jung, and C.-S. Hwang, "Cache strategies for semantic prefetching data," in Web-Age Information Management Workshops, 2006. WAIM'06. Seventh International Conference on. IEEE, 2006, pp. 7–7.

[54] M. Ahmad, S. Asghar, M. A. Qadir, and T. Ali, "Graph based query trimming algorithm for relational data semantic cache," in Proceedings

of the International Conference on Management of Emergent Digital EcoSystems. ACM, 2010, pp. 47–52.

[55] M. Ahmad, M. A. Qadir, and T. Ali, “Indexing for semantic cache to reduce query matching complexity,” Journal of the National Science Foundation of Sri Lanka, vol. 45, no. 1, 2017.