Sub-query Fragmentation for Query Analysis and Data Caching in the Distributed Environment

Santhilata Kuppili Venkata*[1] and Katarzyna Musial†[2]

[1]The National Archives, London, United Kingdom
[2]Advanced Analytics Institute, School of Computer Science, University of Technology Sydney, Sydney, Australia

Abstract

The world of query-response systems heavily depends on the cloud storage solutions, distributed data transfers and locality of users etc. When data stores and users are distributed geographically, it is essential to organize distributed data cache points at ideal locations to minimize data transfers. This leads to the question, *what* data to cache in *which location*. To answer this, we are developing an adaptive distributed data caching framework that can identify suitable data chunks to cache and move across a network of community cache locations. This paper details the first step of the process: the sub-query fragmentation technique to fragment data into portable objects. Evaluation suggests that sub-query fragments enable distributed learning methods to understand query patterns and association between sub-queries. The sub-query objects can be modelled easily as input dataset to implement machine learning models to assist cache maintenance.

Keywords: Query analysis, distributed data caching, query modeling

1 Introduction

Nowadays, it is common for organizations to store their data distributed geographically such as cloud storage systems. Organizations provide a query-response layer over the data storage and hide the location details from users. However, when the responses to user queries (requests) are involved with heavy data transfers, the query analysis provide deep insights into the data placement plans. Clouds and hybrid cloud storage systems are in need of newer technologies to tackle the data placement problem. If the user queries request for a specific piece of data repeatedly, the data item becomes

*santhilata.venkata@nationalarchives.gov.uk - Corresponding author
†kaska@uts.edu.au

a candidate for caching. Since users are also distributed, a data item can be requested from multiple locations. The data item may be requested repeatedly from various locations from different locations. We need to analyse the patterns of these queries to understand the request flow for a cached data item.

In order to answer the above problem, we have developed **CommCache** - a distributed adaptive community caching framework to aid distributed data storage systems. CommCache accepts consists of a central query analyser to accept user queries and fragments them to model them into patterns. Queries are fragmented using sub-query fragmentation technique. In this paper, the sub-query fragmentation technique is detailed to identify the most ideal sub-query fragment for data caching. The sub-queries can be modelled as objects to be placed across network of caches.

Here is a scenario to understand the CommCache environment. It is common for communities of users from different parts of the world to collaborate on projects and access common data stores. It is observed that often their requests to retrieve data partially overlap. It means, it is not necessary that the entire data related to a particular query should be cached. Only a part of the query may be is in common with another query. It follows that certain data segments are frequently needed across different locations at different times, as specified by the data flow pipeline. We utilise this feature to propose sub-query fragmentation for distributed data caching.

When a sub-query q_i of a query Q is a part of several queries, then q_i becomes an good candidate for caching. Ideally each q_i in Q is stored on a cache unit at a location near the data usage. In our research, we aim to achieve the following: (1) model data component(s) as independent distributed objects, (2) define query structures to represent the most frequently searched data and location of the data usage and (3) describe operations for distributed search and retrieval of the cached data. In this paper, we present formal definitions and modeling of data segments and evaluation of the model under various simulated conditions.

Sub-query fragmentation (SQF) [1] defines the process of identifying fragments (sub-queries) of a query execution plan. Sub-query fragmentation follows the rules of semantic caching [2, 3, 4] for initial query fragmentation. Initially, a query plan and result are cached according to semantic caching rules. But over repeated accesses, query plans are fragmented into sub-queries or aggregated and cached as separate queries. Periodically, the cached queries are examined for their frequency of use. SQF differs from other fragmentation techniques in two ways. One, cache units save sub-queries that are easily portable across cache units. When multiple users request the same segment from various geographical locations, transferring a smaller data segment consumes fewer network resources. Two, with smaller chunks, the aggregation process of query results do not need to be performed on the data server. Instead, processing can be delegated to the user location, thus avoiding resource-consuming processing at the data server. It leads to the late binding of partial results to process the query result only when and where it is needed. SQF facilitates a quick lookup of frequent sub-queries and remaining parts of the cached query are evicted when they are not in demand. Distributing smaller segments to caches near user locations provide support for technologies such as Edge caching and Fog computing [5, 6, 7].

2 Background & Related Work

The concept of data caching to reduce response time and volumes of data transfers is highly researched in the past. Multiple experiments with a variety of cache grains are available in the literature. Page caching and tuple caching are and the most widely used approaches to cache results of the query. Page caching caches a collection of index and data pages [8]. Tuple caching caches collections of tuples [8, 9]. Though of tuple caching caches accurate data, the maintenance overhead makes tuple caching a non-feasible technique to implement in distributed applications. Attribute caching is even a fine grained model: it stores data [10] at the attribute level of a table. Unless attribute caches are highly specific to an application, they may not be used for larger tables. Both tuple and attribute caches have very high maintenance for general applications. None of these caching methods is suitable for distributed caching for the overhead to maintain data from multiple databases.

Chen et al. [11] were the pioneers of query result caching and query matching. In their work, authors proposed about intermediate results to cache. They have defined a directed graph whose nodes reference to base relations and cached temporaries and edges represent derivation paths. Their graph representation helps the direction of query fragments and the path to process the data.

Unlike pure result caches, semantic caching [2] and predicate-based caching [3] cache queries along with query results. Queries form metadata along with the records of the data. This approach allows partial matching of a query result even when the cached query results do not satisfy the new query entirely. These methods provide an accurate semantic description of the content of the query. This approach is much better than result caches as the semantic cache allows more queries to reuse the already cached content. It also, allows the formulation of a query to retrieve an exact set of missing result tuples from the server. The semantic caching model introduced concepts called semantic regions: the probe query and the remainder query. Semantic regions within the cache are associated with the collection of tuples. Their replacement policies for maintenance are based on the usage information of semantic regions. A probe query is the part(s) of the query that can be answered by the cache, and the remainder query is the part(s) of the query for which data should be brought from the databases (the cache misses). This initial model of the semantic cache only supports select-project queries but does not support joins in the query. Dar et al. [2] compared their query and result caching with tuple caching, and page caching and show that semantic caching avoids the high overheads of two traditional caching methods. Keller et al. [3] present their model of semantic cache to be able to process select-project-join queries. They allow 'WHERE' conditions and range predicates. This caching scheme is for a central server with multiple clients. They claim lower response times and reduced message traffic, higher server throughput and scalability compared to page-caching. However, semantic caching suffers with the addition of attributes or dropping parts from WHERE conditions, which might increase the resultset size. This approach may waste memory resources which are at a premium in caches. Also, it may add to network communication costs. Another issue with semantic caching is, when a cache entry is not enough to answer a new query, the remainder query must fetch the new data and merge it with the existing cache entry. It may include joining the cache results with a new table, extending ranges of *WHERE* clauses and other constraints. The system must take care such that the remainder query and the merging of the remainder query with the cache entry do not cost more than the original new query. However, if the construction of the remainder query is successful, the semantic caching produces little overhead and reduces response time and network overhead [12]. Semantic caches have caught the interest of many researchers for their ability to identify semantic meaning. In a further development, Ren et al. [4] introduced a formal semantic cache model for select-project queries and single relations. They explained coalescence and decomposition to avoid redundant data. To use semantic caching in the distributed environment, Ryeng et al. [13] proposed a globally distributed cache based on autonomous sites. They have built caches for intermediate results on the sites where the results are produced. In their design, subsequent similar queries can benefit from the stored caches. Semantic caching has been applied to deductive databases [14], federated databases [15], web sources [16, 17], and web querying systems [18]. They are all built for a single entry point to the system.

Since semantic caching creates a region of interest, it makes sense to cache a join of two tables rather than individual tables themselves. With semantic caching, it is possible to cache query plans and collect statistics for the use of join product of tables rather than tables. In general, semantic caching proves to be an ideal candidate to extend as it keeps the semantic record of a query. However, semantic cache seems to be ill-suited for distributed databases due to the remainder query merging. We need to address this issue and find a suitable way to make it applicable to distributed databases. Though semantic caching can be successfully applied for middleware caching [13], it needs to be extended to match the user query request patterns and cache data placement. A distributed cache model needs to be defined for content caching (query and result caching).

Another stream of caching is to cache query execution plans. Instead of caching an entire dataset, smaller join resultsets from execution plans are identified to cache. Since query execution plans depending on the locality of the data, execution plan caching is more suitable for distributed data caching. Rao et al.[19], proposed the invariant technique for correlated queries. In nested queries, sub-queries that are not related to the external references are cached for reuse. We use the idea of independent sub-query or invariants. Our model uses query execution tree to obtain the sequence of sub-queries to identify the most used part of the query. Zanfaly et al. [20] performed analysis of multi-level caching of query plans in distributed databases. They have published the distributed model of query caching at pre-defined locations. This model does not include the performance for changing workloads for changing user needs. A detailed study about query language, access methods, cost-based query optimiser are presented for the structured and semi-structured data [21].

In other types of caching for distributed caching, the object caching method is proposed for data-intensive applications by Haas et al.[22]. Their approach was to load the cache with relevant objects as a by-product of query processing. Due to the applicability with objects, object caching seems to be an ideal way [23] to implement with middleware systems to access non-relational databases. Object caching gains by having fewer faults by eliminating the need to fragment in objects. The object-based approach seems to be a way to cache contents on the distributed systems. However, object caching suffers from an increase in the cost of queries incurred by storing extra data. Also, queries may need multiple objects. Hence, the overhead incurred with object-level caching may be more than query caching. Many studies have been carried out on distributed multi-query processing by [24, 25, 26] using active semantic caching. Andrade et al. [24, 25] studied the benefit through cached results in the proxy either directly or through transformations to the results. They exploited processing commonality across a set of concurrently executing queries and reduce execution times by using previously computed results with an objective to produce better query optimisation.

In an extension, D'Orazio et al. [26] have proposed distributed query scheduling policies in the grid environment. These policies directly consider the available cache contents by employing distributed multidimensional indexing structures and an exponential moving average approach to predict future cache contents. While the aim to re-use partial query results is similar to us, we differ in approach to developing sub-queries. They identify parts of a semantic query and use the coordinates for indexing. Our approach is to divide a user query into sub-queries based on the query plans. Our method allows sub-queries to associate with others when they are repeatedly requested together. The advantage of sub-query fragmentation approach is, it is possible to transfer the sub-query (as a part of pre-placing) when needed at other cache locations. Our approach makes query caching suitable for distributed computing.

3 CommCache - Distributed Cache Environment

This section details the key architectural features for query analysis of CommCache with the help of examples. In its current version, CommCache is implemented with a global centralised environment for query processing. We consider a network of distributed caches connected to the central query processing unit. As shown in Figure 1, user queries are submitted to the central query analyser. The query processor receives user queries and makes query execution plans.

The working of CommCache is divided into two phases; (i) the *active cache phase* during which, the cache management collects queries and query patterns are analysed and (ii) the *cache maintenance phase*, the cache replacement policies are chosen to maintain the cache coherence and re-usability of cached content. Both these phases complement each other to decide **what** to cache, **how long** to cache and **where** to cache the data. The query analyser collates query plans, fragments them into sub-queries and assumes the responsibility for searching and retrieval of all fragments available for a query during the active cache phase. During the maintenance phase, it examines semantic regions of interest and relocates sub-queries according to demographic information, if necessary. Thus, SQF tracks changing workloads and readjusts the cache contents.

In our simulations, the query workloads are assumed to be a continuous stream of queries from users to distributed data stores. Throughout this paper, the concepts and notions introduced will be illustrated by means of two scenarios with sample queries.

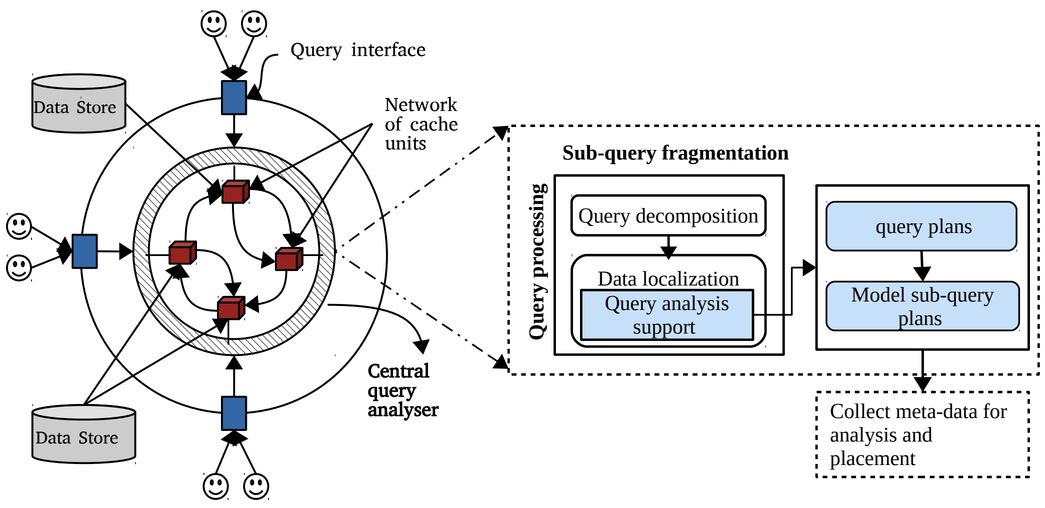


Figure 1: Sub-query fragmentation unit in the CommCache environment

Scenario 1. There are two non-replicated databases (DB1 and DB2) at two locations (separated geographically), *Location1* & *Location2* respectively. Tables *employee(empId, name, age)*, *project(projId, projName, empId)* are part of *DB1* and table *estimation(projId, projLoc, cost)* is a part of *DB2*. The database instance is shown in the Figure 2. These databases do not need be homogeneous databases. We assume translation of data across heterogeneous databases to be an implicit step during the query planning. Consider queries, query1, query2 and query3 that request data from these databases.

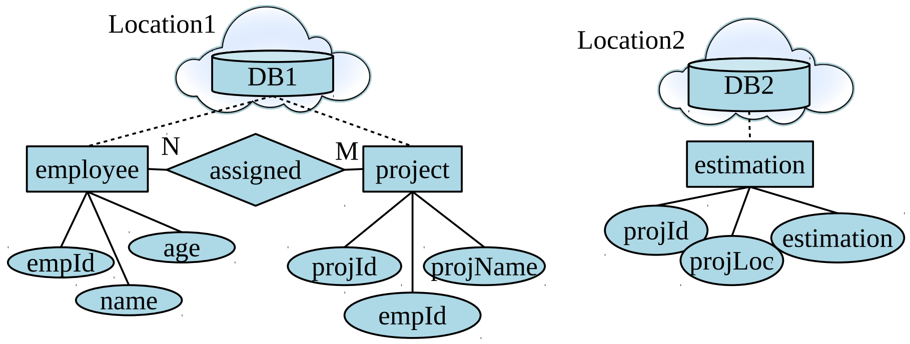


Figure 2: Two databases and their contents used in Scenario 1

Query 1 *This query is to find names of employees, their projects and the project locations, where project cost is under 50,000.*

```
SELECT empName, projName, projLoc
FROM employee(DB1), project(DB1), estimation(DB2)
WHERE ((employee.empId=project.empId) AND
(project.projId=estimation.projId) AND
(estimation.cost < 50000))
```

Query 2 *This query obtains the names of projects that cost under 50,000 where employees older than 45 years of age are working.*

```
SELECT projName
FROM employee(DB1), project(DB1), estimation(DB2)
WHERE ((employee.age > 45)AND((employee.empId = project.empId)
AND (estimation.cost < 50000)))
```

Query 3 *This query is to find all employee names who are older than 45 years.*

```
SELECT empName
FROM employee(DB1)
WHERE (employee.age > 45)
```

Scenario 2. Two horizontally fragmented distributed data centres contain horizon-

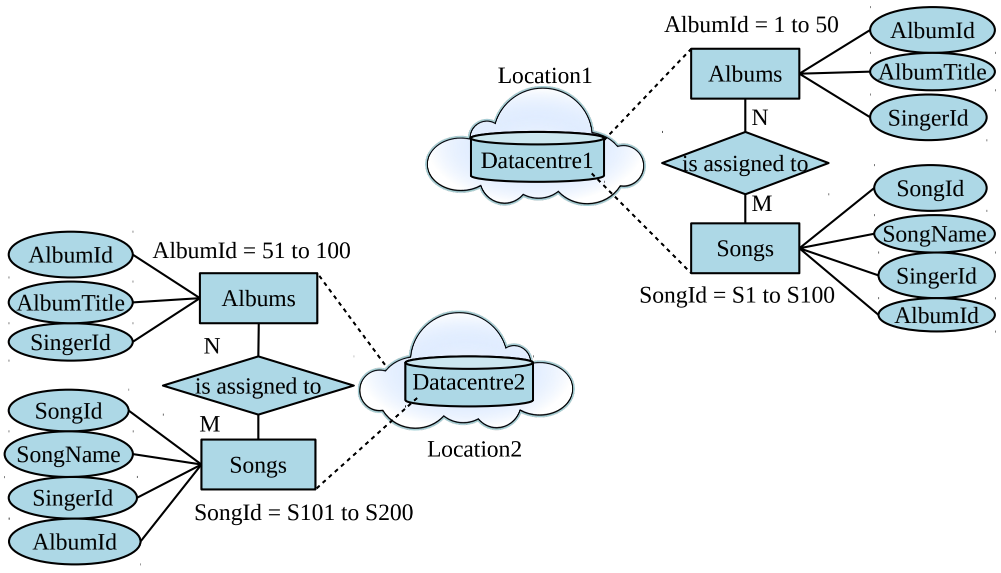


Figure 3: Model of horizontally fragmented distributed data centres in Scenario 2.

tally fragmented data of two tables *Albums(AlbumId, AlbumTitle, SingerId)* and *Songs (SongId, SongName, SingerId)* at two geographically separated locations. The table 'Albums' is horizontally fragmented on the primary key *'AlbumId'*. Similarly, the table *'Songs'* is horizontally fragmented on the primary key *'SongId'*.

All tuples in *Albums* table ranging from 1 to 50 are stored at Location1 in *Datastore1* and tuples from 51 to 100 are stored at *Location2* in *Datastore2*. All tuples in the *Songs* table ranging from S1 to S100 are stored at *Location1* in *Datastore1* and the tuples from S101 to S200 are stored at *Location2* in *Datastore2*. Database instance is as shown in Figure 3. Consider the following queries sent to these data centres[1].

Query 4 *This query obtains all song names stored in the table* Songs *from all fragments of the table.*

```
SELECT SongName FROM Songs;
```

[1]This scenario is a reproduced version of the example given at https://cloud.google.com/spanner/docs/query-execution-plans

Query 5 *This query is to get titles of albums and names of songs sung by singers present in albums.*

```
SELECT al.AlbumTitle, so.SongName
FROM Albums AS al, Songs AS so
WHERE al.SingerId=so.SingerId AND al.AlbumId = so.AlbumId
```

4 Sub-Query Fragmentation

The previous section introduced the environment within which dispersed groups of users query distributed databases. This section introduces the theoretical model of sub-query fragmentation that enables the distributed caching system that serves those groups of users.

4.1 The Sub-query Concept

We can formally define a sub-query as

Definition 1 *A query q_i is a sub-query of a query Q if q_i is a valid query that can be executed independently and its result can be combined with that of other sub-queries of Q to produce the result of Q.*

A sub-query can be further fragmented into one or more sub-query plans depending on the need and possibility to fragment further. Since sub-queries are the parts of query execution plans, an **atomic sub-query** is defined as an indivisible sub-query or the aggregation of sub-queries that are often queried together. One of the possible query plans for this query is given in the Figure 4. In this plan, all the data needed from DB1 is retrieved through a single access. For the purpose of illustration of distributed processing of the query, this plan ignore any local execution plans within a database to compute local results.

Example 1 *Consider the Query 1 from Scenario 1.*

- *sub-query q_{11}:*
  `SELECT employee.empId, project.projName, project.projId`
  `FROM employee, project`
  `WHERE employee.empId = project.empId` $\Rightarrow$ *(partial) local result* (qr_{11})

- *sub-query q_{12}:*
  `SELECT (estimation.projId, estimation.projLoc)`
  `FROM estimation`
  `WHERE (estimation.cost < 50000)` $\Rightarrow$ *(partial) local result* (qr_{12})

- *sub-query q_{13}:*
  `SELECT (empName,projName, projLoc)`
  `FROM` qr_{11}, qr_{12}
  `WHERE (`qr_{11}`.projId = `qr_{12}`.projId)` $\Rightarrow$ *Final result* (QR1)

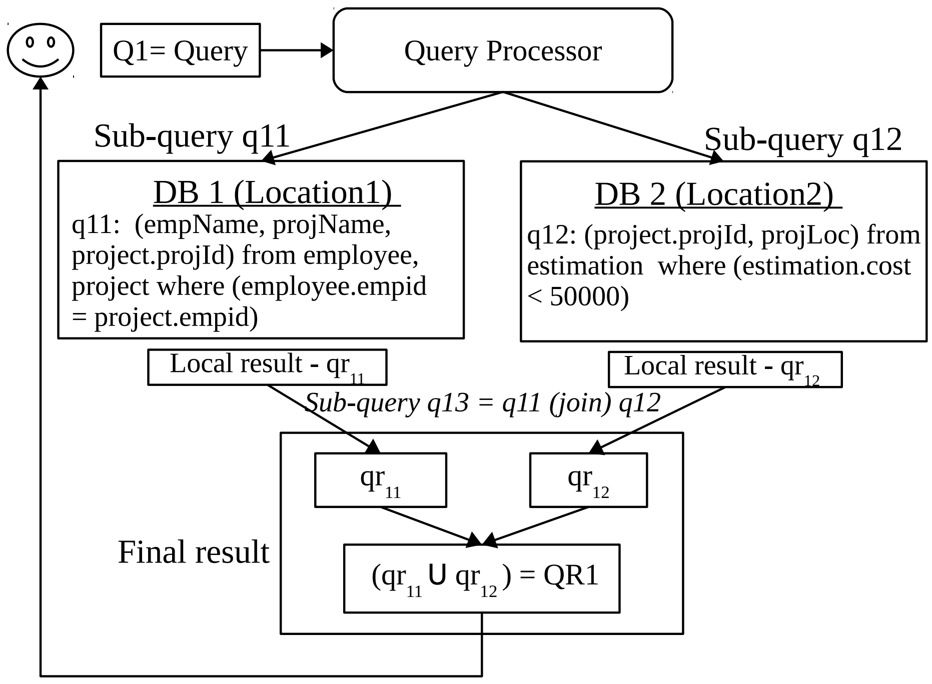

Q1= Query
Query Processor
Sub-query q11
Sub-query q12
DB 1 (Location1)
q11: (empName, projName, project.projId) from employee, project where (employee.empid = project.empid)
DB 2 (Location2)
q12: (project.projId, projLoc) from estimation where (estimation.cost < 50000)
Local result - qr11
Local result - qr12
Sub-query q13 = q11 (join) q12
qr11
qr12
Final result
(qr11 ∪ qr12) = QR1

Figure 4: Sample query plan for Query 1

The sub-query q_{11} has a join operation between tables *employee* & *project*. Sub-query q_{12} is an *atomic* sub-query, as the result data fragment is produced as a single unit. Sub-query q_{13} is a distributed join (or aggregation shown as $\bigcup$ in Figure 4) of the result of two sub-queries q_{11} and q_{12}. Since sub-queries are executed independently, each sub-query can be considered as a query on its own.

In general, a sub-query q_i can be described by a tuple q_i=$\langle {q_i}^R, {q_i}^A, {q_i}^P, {q_i}^C\rangle$ as defined in [4] and is explained in Table 1. Since atomic sub-queries are similar to semantic segments, we can derive that a sub-query s can be answered by t given that $(s^C \wedge t^C) = s^C$. In other words, $s^C \subset t^C$. Similarly, other comparisons such as s^C is *equivalent* to t^C, to determine whether s can be answered by t follow the definitions of [4] while searching the cache for contained query.

4.2 Notational Representation of a Query

A Query Q is an aggregation of its sub-queries executed in a combination of parallel and sequential execution as defined by the query plan. We introduce an execution operator '$\|$' for sub-queries that can be executed concurrently. Similarly, '_' represents a sequential operator for sub-queries that must be executed in a predefined sequence. Thus, a query plan can be written using the infix notation[2] of sequential and parallel

[2]Infix notation is a simple to read notational representation for writing algebraic and logical expressions. Operators are written in-between their operands. Parentheses override operators to resolve precedence and associativity.

Attribute	Description	Example value
${q_i}^R$	The set of relations in the query	employee, project
${q_i}^A$	The set of all attributes to be accessed through select part	employee.empId, project.projName, project.projId, project.empId
${q_i}^P$	The set of predicates	employee.empId=project.empId
${q_i}^C$	The resulting tuples of the query	(qr_{11})

Table 1: Tuple attributes of a sub-query for the sub-query q_{11}.

execution operators written in-between their sub-queries. It follows that,

$$(q_1 \cup q_2 \cup .. \cup q_n) = \bigcup_{k=1}^{n} q_k = Q \tag{1}$$

where, $\cup$ denotes a parallel or sequential execution operator for aggregation.

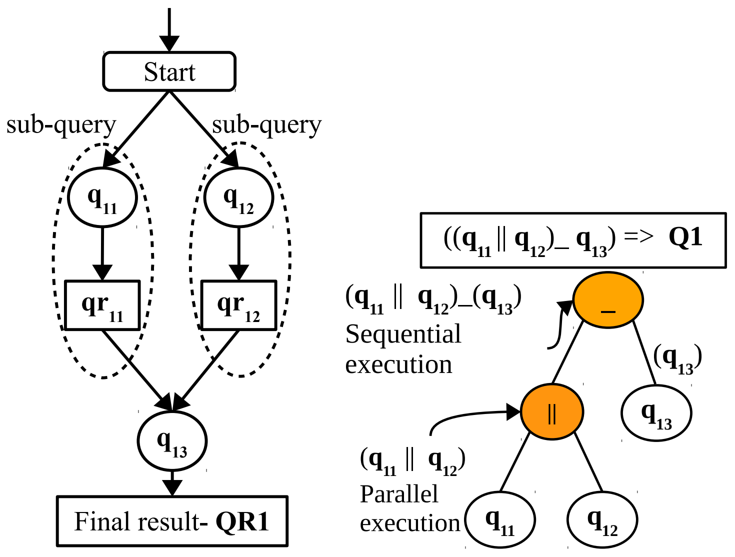
A simplified sequence of operations for sub-query execution to get the query result is shown in Figure 5a for $Q1$. Let the intermediate result generated by the sub-query plan q_{11} be qr_{11} and that of sub-query plan q_{12} be qr_{12}. The overall result is aggregated as $QR1$ on join *(qr_{11}.projId = qr_{12}.projId)*. Sub-query plans q_{11} and q_{12} can be executed as two independent plans in parallel. The aggregation for q_{13} depends on the results of q_{11} and q_{12}. It means, execution of q_{13} cannot be performed until the execution of both q_{11} and q_{12} are completed. Hence q_{13} is executed sequentially after q_{11} and q_{12}. Using the notation, $Q1$ is:

$$(((q_{11}) \parallel (q_{12})) \mathbin{\_} (q_{13})) \Rightarrow Q1 \tag{2}$$

4.3 The Query Evaluation Tree

A query evaluation tree (QET) for a query Q is the graphical representation of a query in the infix order of execution. It is similar to the query execution tree presented in [19]. In a QET, sub-queries are found at leaf nodes. Each intermediate node is an execution operation performed on child nodes. When there is more than one child plan to a parent node, it is assumed that child plans are to be executed in the order of left node to right node. When a sub-query needs to be fragmented further, the evaluation tree replaces an execution operator, and further fragmented sub-queries are added as child nodes.

When a new query appears for the first time, the whole query result is cached as a single node. When parts of the query are requested by other queries, then the original query fragments themselves into sub-queries depending on the demand. Each leaf node of the evaluation tree contains a sub-query plan and address to the physical location of



(a) Query plan with sub-queries

(b) Tree with execution operators

Figure 5: Query Evaluation Tree (QET) for the query $Q1$

the sub-query's result. A leaf node consists of metadata of the sub-query result. The query evaluation tree for the query $Q1$ with execution operators is shown in Figure 5b. The structure of the node is given in Table 2.

Attribute	Value
Sub-query (q_i)	infix expression (Notational representation of q_i)
Semantics (q_i)	$\langle {q_i}^R, {q_i}^A, {q_i}^P, {q_i}^C \rangle$ (Description of semantic segment)
Address	a pointer to the physical address of q_i

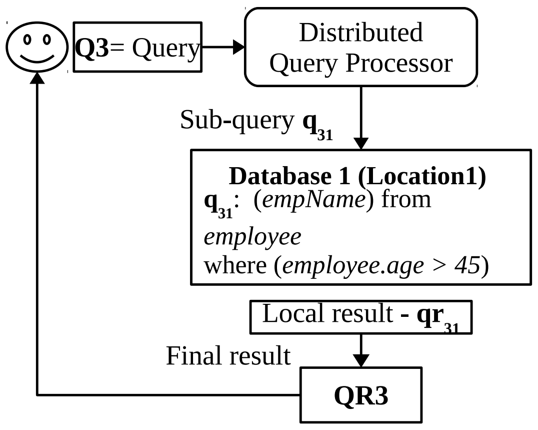
Table 2: The structure of a node in the query evaluation tree

4.4 Query Complexity

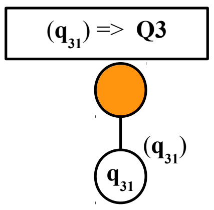
The Complexity of a query is the number of sub-queries to be executed in a query plan. In other words, complexity is the number of leaf nodes of the query evaluation tree. The complexity of a query is used to understand the processing requirements for a given query. A higher number of sub-queries in a query indicates higher processing overheads. Query complexity influences outcome of the analysis of query patterns. It is used in the decision making whether it is optimal to fragment a query further during the maintenance phase.

Example. Consider the query 3 from Scenario 1. This query is to find all employee names who are above 45 years of age.

```
SELECT (empName)
FROM employee(Database1)
WHERE (employee.age > 45)
```


Q3= Query
Distributed Query Processor
Sub-query q_{31}
Database 1 (Location1)
q_{31}: (*empName*) from *employee* where (*employee.age* > *45*)
Local result - qr_{31}
Final result
QR3

(a) Sample query plan for query 3



(b) Tree with execution operators

Figure 6: Query Evaluation Tree (QET) for the query $Q3$

For this query, a possible query plan is shown in Figure 6a with a single sub-query. The equivalent query evaluation tree is shown on Figure 6b.

- sub-query q_{31}: `SELECT empName FROM employee(DB1) WHERE (employee.age>45)`

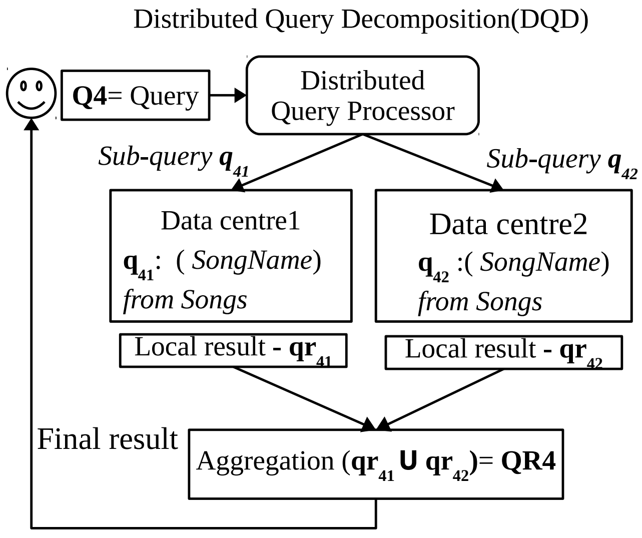
Example. Consider query 4 from Scenario 2.
`SELECT SongName FROM Songs;`
For this query, a possible query plan is shown in Figure 7a with following sub-queries:

- sub-query q_{41}: `SELECT SongName FROM Songs (Location 1)`
  (This sub-query retrieves all *SongNames* from *SongId=1* to *SongId=100* from the horizontal fragment at Location 1)

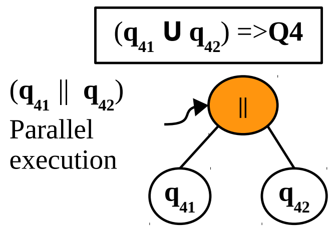
- sub-query q_{42}: `SELECT SongName FROM Songs (Location 2)`
  (This sub-query retrieves all *names of songs* from *SongId=101* to *SongId=200* from the Horizontal fragment at Location 2)

- final result: `Aggregation of results of sub-query` q_{41} `and sub-query` q_{42}

Since this query has to access data from two fragments of the table *Songs* from two locations, it is convenient to consider them as two separate sub-queries from the view of network resources. Though the query leads to a semantically single query, it is advantageous to retrieve and cache data from these fragments separately for the sake of the query execution. Similarly, we consider them as two sub-queries even when the data comes from two replicas of the same table. (As replication is done to do load


Distributed Query Decomposition(DQD)
Q4= Query
Distributed Query Processor
Sub-query q41
Sub-query q42
Data centre1
q41: (SongName) from Songs
Data centre2
q42 :(SongName) from Songs
Local result - qr41
Local result - qr42
Final result
Aggregation (qr41 U qr42)= QR4

(a) Sample query plan for query 4



(b) Tree with execution operators

Figure 7: Query Evaluation Tree (QET) for the query $Q4$

balancing across servers, it is logical to consider them as two separate data centres). The third operation is only the aggregation of two query results. Since it is a simple union operation performed on two sub-query resultsets we do not consider as a separate sub-query. This query has two sub-queries. Hence the complexity of the query 4 is 2. Similarly, the query 1 (in Figure 5b) has three leaf nodes. So its complexity is 3.

4.5 The Cached Query Model

When a frequent sub-query is cached, it becomes an independent *cached query object*. We use a '$\Diamond$' notation to distinguish a cached query '$\Diamond S$' from a user query S.

A cached query is made up of two parts. One, that defines the sub-query plan and second, a pointer to the physical address where the result is cached. The object profile of a cached query consists of the usage information of the query during its 'cache life time[3]'. These attributes provide adequate information for the analysis and prediction of future needs of users during cache maintenance.

A **Cached Query** $\Diamond S$ is a sub-query result and along with its meta-data parameters. A cached query is represented as a tuple
$\langle \Diamond S, CLoc, V, C, \{T_{uLoc}\}, \{F_{uLoc}\}, \{D_i\}\rangle$.

Example. A cached query $\Diamond S =$
$\langle((q_a \parallel q_m)\_q_n)$, (cache-2), 10, 3, {3,4,2}, {10,12,10}, $\{q_a, q_b, q_k\}\rangle$.
The description of these parameters is explained in Table 3.

[3]The lifetime is the length of time a sub-query is stored in the cache

Attribute	Description	Value
$\lozenge S$	The query expression of S	$((q_a \parallel q_m)\_(q_n))$
$CLoc$	Location of the cache server, where $\lozenge S$ is currently cached	(*cache-2*)
V	Volume of the cached data of $\lozenge S$ in GB	10
C	Complexity of $\lozenge S$	3
$\{T_{uLoc}\}$	A set of time stamps, $\lozenge S$ recently used at each of the user locations 'uLoc'	{(*uloc-1*),3}, {(*uloc-2*),4}, {(*uloc-3*),2}
$\{F_{uLoc}\}$	A set of frequencies (no.of times query accessed) $\lozenge S$ has been requested from each user location 'uLoc'	{(*uloc-1*),10}, {(*uloc-2*),12}, {(*uloc-3*),10}
$\{D\}$	List of other queries with which $\lozenge S$ has been queried together	$\{q_a, q_b, q_k\}$

Table 3: Description of a Cached Query $\lozenge S$

4.6 Cache Granularity

The *cache granularity* is the smallest independent cacheable object. For an attribute or tuple caching policy, the granule is an attribute or a tuple respectively. Commonly, data transfers are measured as the number of memory pages [27]. The physical size of a cache granule is a page. Though query objects are cached at the page level, from the logical point of view, a sub-query (result) is the granule with sub-query fragmentation. A SQF cache granule can have a variable size due to the frequency of a sub-query requested along with other sub-queries.

4.7 Query Equivalence and Overlap

The query equivalence is essential when checking whether a user query can be answered by the cache. Equivalence checks whether the result needed by a user query is equivalent to, or part of, any of the cached queries.

4.7.1 Equivalent Queries

Definition. Two queries $(S, \lozenge T)$ are **equivalent** $S \equiv \lozenge T$, when their query evaluation trees have identical internal nodes and the participating sub-queries are equivalent. It follows that $(S \cap \lozenge T) = S$ or $\lozenge T$.

Definition. Alternately, two queries $(S, \Diamond T)$ are **equivalent** $S \equiv \Diamond T$, when their final results yield the same resultset with different sub-queries.

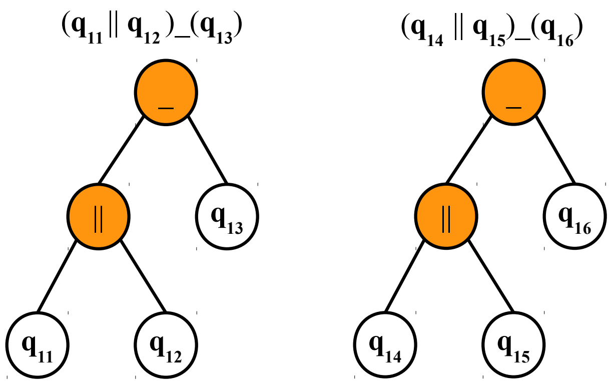
Since the root node of a query evaluation tree is an aggregation of the complete query, equivalence for two queries can be checked by the query expression or answerable by the semantic definition [4] (explained in the Table 2) at its root node.

Example. Distributed query plan generation depends on many factors such as load distribution, process distribution, and location dependence of data availability [28]. A query may have more than one execution plan leading to many possible strategies. For example, another query execution plan for Query 1 (from Scenario 1) can be with following sub-queries:

- *sub-query q_{14}:* `SELECT empId,empName`
  `FROM employee`⇒`partial result (`qr_{14}`).`
- *sub-query q_{15}:*
  ```
  SELECT project.projName, project.empId, estimation.projLoc
  FROM project, estimation
  WHERE (project.projId=estimation.projId) AND
  (estimation.cost < 50000)⇒partial result (qr15).
  ```
- *sub-query q_{16}:* `SELECT empName, projName, projLoc`
  `FROM `qr_{14}`, `qr_{15}
  `WHERE ( `qr_{14}`.empId = `qr_{15}`.empId )`⇒`complete result (QR1).`

The complete query plan with the above sub-queries is

$$(((q_{14}) \parallel (q_{15})) \_ (q_{16})) \Rightarrow Q1 \tag{3}$$



(a) Execution plan 1 (b) Execution plan 2

Figure 8: Equivalent query evaluation trees for Q_1 with different sub-queries

Two query evaluation trees for Q_1 with sub-queries q_{11}, q_{12}, q_{13} and q_{14}, q_{15}, q_{16} are shown in 8a and 8b respectively in the Figure 8. Though, the leaf nodes of the above evaluation trees are different, the semantic information at the root nodes yields the same resultset, both trees are considered to be *equivalent*. Algorithm to check the equivalence of two queries is given in Algorithm 1.

Algorithm 1 EquivalentQuery (S,T)

```
1: Input: Query $S$, Query $T$
2: Output: boolean (true or false)
3: root-S $\Leftarrow$ root of QET for $S$                ▷ root node of the cached query
4: root-T $\Leftarrow$ root of QET for $T$                ▷ root node of the query to search
5: if (root-S.sub-query equals root-T.sub-query) then
6:     return true
7: else return false
8: end if
```

4.7.2 Query Overlap

Definition. Query S is **completely overlapped** by a query $\Diamond T$, i.e., ($S \subseteq \Diamond T$), when the *root node* of query S is **equivalent** to *one of the nodes* of the (query $\Diamond T$). It follows that, $(S \cap \Diamond T) = S$.

Example. Consider Query 2 and Query 3 from Scenario 1:
Query 2 obtains the names of projects (with cost < 50000) where employees above the age of 45 years are working.

```
SELECT (projName)
FROM employee(DB1), project(DB1), estimation(DB2)
WHERE ((employee.age>45) AND ((employee.empId=project.empId)
AND (estimation.cost < 50000)));
```

Query 3 is to find all employee names whose are above 45 years of age.

```
SELECT (empName)
FROM employee(DB1)
WHERE (employee.age > 45)
```

In the above queries, Query 3 is completely overlapped by Query 2.

Definition. Query S and query $\Diamond T$ are **partially overlapped**, when *one or more nodes* of the evaluation tree of S are **equivalent** to *one or more nodes* of the evaluation tree of query $\Diamond T$. It follows that, $S \cap \Diamond T \neq$ *null*.

Example. Consider sub-queries of Q_1 and Q_2 of Scenario 1.

- *sub-query q_{11}:*
  ```
  SELECT employee.empId, project.projName, project.projId
  FROM employee, project
  WHERE employee.empId=project.empId
  ```
- *sub-query q_{12}:*
  ```
  SELECT estimation.projId, estimation.projLoc
  FROM estimation
  WHERE estimation.cost < 50000
  ```
- *sub-query q_{13}:*
  ```
  SELECT empName, projName, projLoc
  ```

```
FROM qr11, qr12
WHERE (qr11.projId=qr12.projId )⇒complete result(QR1)
```

Following are one of the possible query execution plans for Query 2:

- sub-query q_{21}: `SELECT project.projName, project.projId`
  `FROM employee, project`
  `WHERE (employee.empId=project.empId) AND (employee.age>45)`
- sub-query q_{22}: `SELECT estimation.projId`
  `FROM estimation`
  `WHERE (estimation.cost < 50000)`
- sub-query q_{23}: `SELECT projName`
  `FROM qr21, qr22`
  `WHERE (qr21.projId = qr12.projId) ⇒ complete result(QR2)`

From the above queries, the *sub-query* q_{22} can be answered by *sub-query* q_{12}. If the results of query $Q1$ were cached, then the query $Q2$ can be partially answered by the query $Q1$. The remaining part of the query $Q2$, *sub-query* q_{21} cannot be answered by the query $Q1$.

4.8 Searching for Contained Query

The process of searching whether a query contains the result is done by checking the equivalence. Since we consider only sub-queries, we introduce a logical layer (such as an index) between the query interface and the physical storage location. The index consists of evaluation trees ordered by the demand for a sub-query. A user query should be searched against each of the root nodes of the cached queries to find what part of the user query can be answered by each of the cached queries.

When a cached query ($\Diamond T$) contains the user query (S), *i.e.*, S is partially or fully overlapped by $\Diamond T$, it is enough to search the query evaluation tree of $\Diamond T$ according to breadth-first order since sub-queries are stored in a top-down manner to get the biggest part that can be answered by $\Diamond T$. We say, set of all sub-queries of S that can be answered by $\Diamond T$ is the **ContainedQuery** ($\blacktriangledown S$), and set of all sub-queries that cannot be answered is the **RemainderQuery** ($\triangledown S$). Algorithm to search for a completely overlapped is present in Algorithm 2. Algorithm 3 explains to search cache for a user query S in general.

$$\text{ContainedQuery } (\blacktriangledown S) = \{s_i \mid s_i \subseteq \Diamond T\} \tag{4}$$

$$\text{RemainderQuery } (\triangledown S) = \{s_i \mid s_i \in (S - \blacktriangledown S)\} \tag{5}$$

From these definitions, we can derive how a query can be answered by the cache:

- A query S can be **completely answered** by the $\Diamond T$ in cache if, $S \equiv \Diamond T$ or $S \subset \Diamond T$.

- A query S can also be **completely answered** by the cache, if all nodes in S can be completely answered by one or more queries in the cache ($S = \blacktriangledown S$) and ($\triangledown S = \varnothing$ (null)).
- A query S can be **partially answered** by the cache, if the RemainderQuery ($\triangledown S \neq \varnothing$ (null)).
- A query S is **not found** in the cache if, ($S = \triangledown S$).

Algorithm 2 isContained ($S \subseteq \Diamond T$)

```
1:  Input: $S$
2:  Output: Internal node ($t_i$ )
3:  if EquivalentQuery($S$,$T$) then
4:      return root-T                                      ▷ equivalent trees
5:  else do Breadth-First-Traversal (root-T)
6:      if root-S equals internal-node ($t_i$) then
7:          return $t_i$
8:      else return null
9:      end if
10: end if
```

Example. Suppose we have a cache with 3 queries stored. A logical view of the cache is shown in Figure 9. Consider a set of queries cached and set of user queries given below.

Cached queries	User queries
$\Diamond T_1 \Leftarrow$ ((q1 || q2) ₋ (q3))	$S_1 \Leftarrow$ (q1 || q2)
$\Diamond T_2 \Leftarrow$ ((q4) || (q5) || (q6))	$S_2 \Leftarrow$ ((q3) ₋ (q4))
$\Diamond T_3 \Leftarrow$ ((q6) || (q9))	$S_3 \Leftarrow$ ((q9) || (q8))
	$S_4 \Leftarrow$ ((q7) ₋ (q8))

The logical view of contained and remainder queries for these user queries are shown in Figure 10. In this example, S_1 can be fully answered by the cache as $S_1 \equiv \Diamond T_1$. Similarly, S_2 can also be fully answered by the cache as the leaf nodes (q3) and (q4) of $S2$ are partially contained in $\Diamond T_1$ and $\Diamond T_2$. However, S_3 can be only partially answered by the cache. It has a remainder query $\triangledown S_3$ = (q8). The contained query $\blacktriangledown S_3$= q9. S_4 cannot be answered at all by the cache as (q7) and (q8) are not stored in cache. Hence, $\blacktriangledown S_4 = \varnothing$ and $\triangledown S_4 = S_4$.

5 Evaluation

The overall performance of a cache system depends on active cache phase and maintenance phase together. Since the sub-query fragmentation is developed for the distributed cache scenario, we compare and observe the significant improvements using this technique with the baseline full query caching and distributed semantic caching in distributed environments.

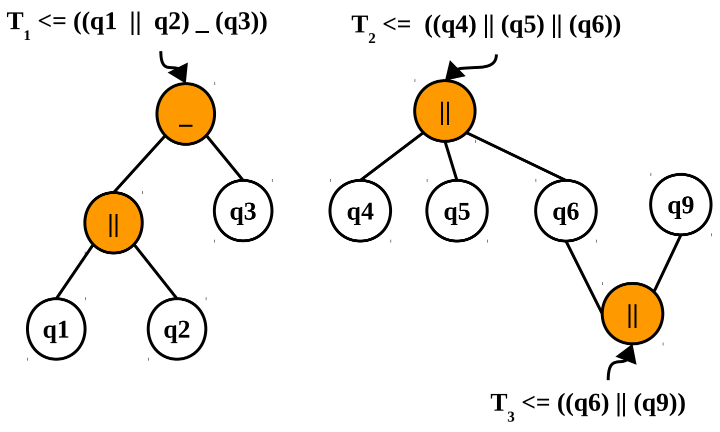


Figure 9: Sample queries present in cache

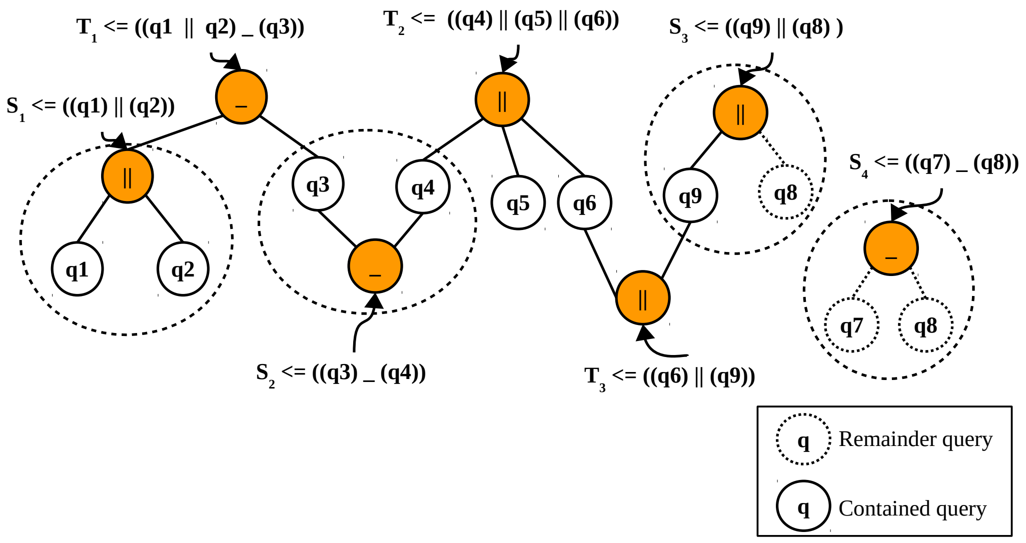


Figure 10: Contained queries and remainder queries

Algorithm 3 SearchCache(S)

```
1:  **Input:** Query $S$
2:  **Output:** ContainedQuery ($\blacktriangledown S$), RemainderQuery ($\triangledown S$) , status
3:  $\blacktriangledown S = \varnothing$
4:  $\triangledown S = \{s_i \mid s_i \in S\}$
5:  **for** each cache unit **do**
6:      **for** each cached query $\lozenge T$ **do**
          ($\blacktriangledown S$, $\triangledown S$) = QUERYSEARCH($\blacktriangledown S$, $\triangledown S$, $\lozenge T$ )
7:          **if** ($\triangledown S = \varnothing$) **then**
8:              return ($\blacktriangledown S$, $\varnothing$, *fully found*)
9:          **end if**
10:     **end for**
11: **end for**
12: **if** ($\triangledown S$ = $S$ ) **then**
13:     return ($\varnothing$, $\triangledown S$, *Not found*)
14: **else** return ($\blacktriangledown S$, $\triangledown S$ , *partially found* )
15: **end if**
16:
17: %Function to search for a query $S$ within a cached query $\lozenge T$
18: **function** QUERYSEARCH($\blacktriangledown S$, $\triangledown S$, $\lozenge T$ )
19:     **Output:** ContainedQuery ($\blacktriangledown S$), RemainderQuery ($\triangledown S$)
20:     **for** each $s_i \in \triangledown S$ **do**
21:         **if** ($s_i \equiv \lozenge T$) OR ($s_i \subseteq \lozenge T$) **then**
22:             $\blacktriangledown S \rightarrow \blacktriangledown S \cup s_i$
23:         **else** $\triangledown S \rightarrow \triangledown S \cup s_i$
24:         **end if**
25:     **end for**
26:     return ($\blacktriangledown S$,$\triangledown S$)
27: **end function**
```

Query Workloads

The query workloads are a continuous stream of queries submitted. Queries access data from multiple databases.

sub-query fragmentation needs constantly changing workloads with partial overlaps. We have developed a synthetic query generator *Qgene* - to generate workloads as query plans with the details of user location(s), cache location and the timestamp.[4]. Qgene allows users to set configuration settings for (i) the duration of the observation time window, (ii) number of queries in the workload per window, (iii) varous statistical distributions for sub-query overlap and inter-query arrival time. Qgene also allows vary query complexity (number of sub-queries). A random factor is introduced to the workloads to eliminate any bias. Each of the following experiments was executed multiple times and obtained an average value.

Experimental Settings

All experiments for the evaluation are conducted on Java based simulator (JDK 1.8). All experiments were conducted for Poisson, Uniform and Exponential distributions of workloads for query overlap and sub-query repetition.

Other variable settings relate to the cache environment. The maximum number of cache agents is decided based on the inter-cache communications that can be handled by the hardware configuration in the laboratory. We set the number of cache units in the network to be 20. In the distributed environment, the ideal data placement algorithm to place data near users influence the response time. We set the data placement to follow Greedy placement policy for a uniform evaluation of the caching techniques.

To understand the impact of cached segments as distributed independent objects and the need for SQF, the caching techniques are evaluated and compared for three metrics; (i) average response time, (ii) cache utilisation and (iii) cache stability. The response time depends on several factors such as (a) caching policies used, (b) cache replacement heuristics, and the (c) distributed data placement methods etc.. Hence, in a fundamental scenario without data placement, SQF and Semantic caching are expected to perform almost the same. Cache utilisation and cache stability metrics actually highlight the advantage of SQF. Each of these metrics are observed for 14 continuous time epochs. Observations for every epoch were repeated for 8 to 12 times. The average is plotted with standard errors.

5.1 Average Response Time

The response time is measured as the time elapsed from a query sent from the user to the time the reply is received. It is a relevant metric as the objective for any caching technique is to minimise the response time. For continuous query workloads, we consider the average response time per workload. In the simulation environment, the response time is measured as logical clock ticks. The symbol notation to calculate average response time lapsed is presented in the Table 4.

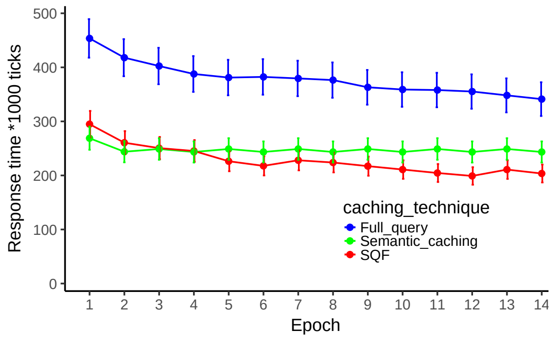
[4]A detailed description of design and query modeling parameters will be provided for evaluation and use on request.

Notation	Description
N	total number of queries
l_t	average cache latency
D_Q	processing time for a query Q at data servers
Q_{proc}	average processing time for a query Q
t_{qi}	data transfer time on network for q_i from cache
$V_Q^{notfound}$	volume of data not found in cache

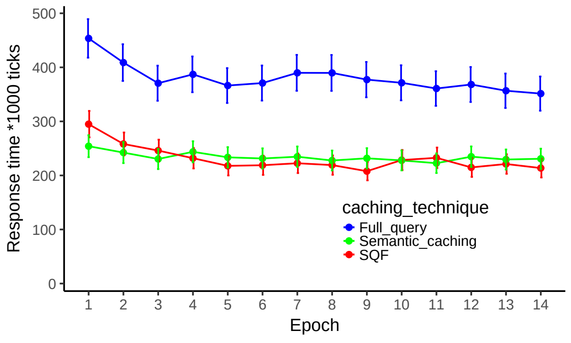
Table 4: Notation Table

$$\text{Average response time} = \frac{1}{N}(l_t + D_Q + t_Q + Q_{proc} + t_{qi}) \tag{6}$$

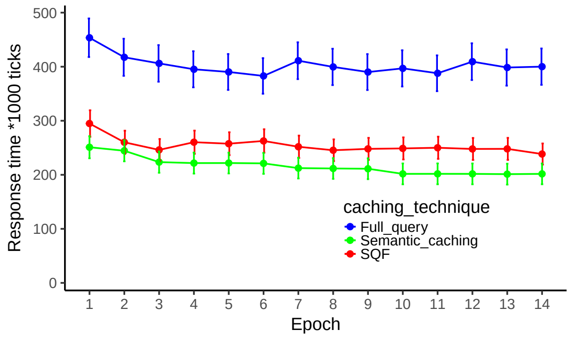
Where,
l_t = query lookup time + actual data retrieval time
D_Q = $V_Q^{notfound} * D_t$
t_Q = time to transfer $V_Q^{notfound} * d_{net}$
t_{qi} = Inter cache transfer time * number of hops



(a) Poisson query input



(b) Exponential query input



(c) Uniform query input

Figure 11: Response time for static workloads

Discussion:

The response time is observed for continuous epochs is plotted in Figures 11a, 11b, and 11c. The full-query resultset caching model stores results as a single unit at one location. Hence it resulted in a high number of cache faults and high response time. In general, this policy observes high response time for all types of query distributions. SQF and SC models performed almost similarly for Exponential and Poisson distributions. Under similar conditions SQF follows semantic rules for the fragmentation of queries. The semantic model showed up to 10% lower response times than SQF for Uniform distribution. One reason for SQF to have high response time could be that SQF further fragments queries according to user requirements and association with other queries in caches. Since sub-queries are repeated uniformly, the overall response time is slightly higher for SQF.

5.2 Adaptivity to Changing Workloads

The plot in the Figure 12 shows the adaptivity of caching techniques to changing query inputs over a continuous period. The input query distribution patterns are changed after every five epochs. The experimental settings are similar to the above experiment. Query workloads are mixed distribution of sub-query repetition.

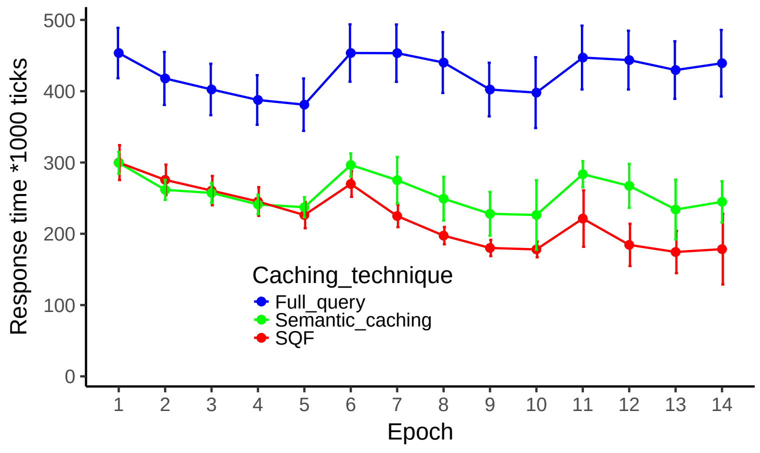


Figure 12: Adaptivity to changing workloads

Discussion:

A sudden spike in response time is observed after fifth and tenth epochs due to the change introduced in the query input. Though, SQF and semantic models responded similarly for the first five epochs, as SQF caches highly associated fragments as a grain, it adapted much better to the variation in the workload than semantic caching or full query model. The advantage of SQF is even more evident with subsequent epochs. This shows that SQF is more powerful than other state-of-art techniques in the distributed environment where the environment is highly dynamic and workloads are continuously changing as it happens in the real-world applications.

5.3 Cache Utilisation - Data Found Vs Caching Policies

The percentage of data found in the cache influences the response time. Higher volumes of data found in the cache (V_Q^{found}) reflect the ability of cache selection to maximise the cache utilisation. In this experiment, we allowed cache models to learn from query patterns. The learning is done by finding out the finest grain of query fragments that are repeated more than a threshold. The learning gets the support from the associated sub-queries with each of the frequented sub-query. The percentage of data found in cache is compared across three caching policies.

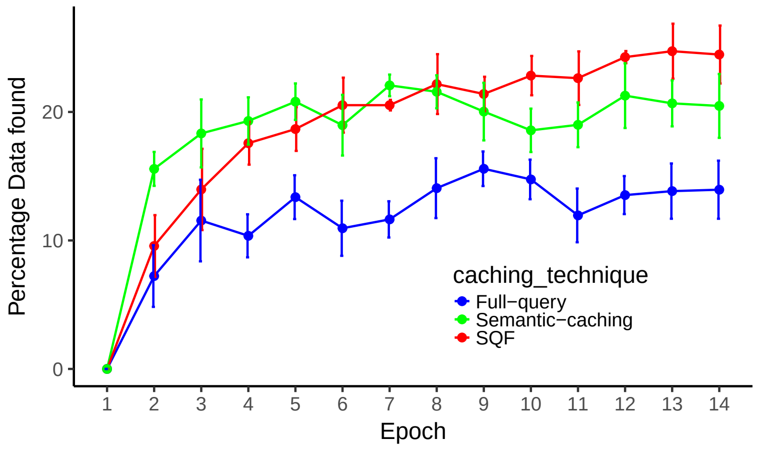


Figure 13: Comparison percentage data found in cache across caching techniques

Discussion:

It is observed from the Figure 13 that the semantic model performed better than other approaches during the initial epochs. Although initial performance of SQF in terms of percentage of data found in cache was lower than this of SC, SQF observed the finer grains of query fragment repetition and cached the query fragments effectively. The observation capability provided SQF to "learn" user requirements up to the finest grain and performed better over the subsequent epochs. Nearly 12.5% more data is found than in the case of semantic caching model. The results in Figure 13 are presented for Uniform distribution.

5.4 Cache Stability - Inter-Cache Data Transfers

The stability of a cache refers to the ability to predict future needs and cache appropriate data. The cache stability measures the utilisation of cache. Higher utilisation of cache content refers to better performance. In the distributed cache environment, stability is a representation of (i) fewer number of relocations of the cached data, (ii) high content re-use and (iii) higher prediction accuracy. Since query and data access patterns often exhibit locality of reference (temporal and spatial locality), data should be cached near the user location where the data has been requested recently (for temporal locality) or stored close to the location of the user (for spatial locality). Let,

Number of sub-query objects to relocate = δ_n
Average volume of a sub-query (q_i) in GB = v_{qi}
Average data transfer cost for inter-cache transfers per GB= d_{net}

$$\text{Average cost Inter-cache data transfers} = \frac{1}{\delta_n} \sum_{i=1}^{\delta_n} (\delta_n * v_{qi} * d_{net}) \tag{7}$$

The parameter (δ_n) varies with the efficiency of the data placement algorithm to place a data segment. The parameter (v_{qi}) varies with changing workloads. Queries with higher complexity lead to higher number of data accesses and hence transfers. The following experiment compares the response time for inter-cache data transfers to reach to the cache set nearer to the user. The comparison is made using three cache techniques. (i) caching with sub-query fragmentation (SQF), (ii) semantically distributed data and (iii) caching full-query resultsets.

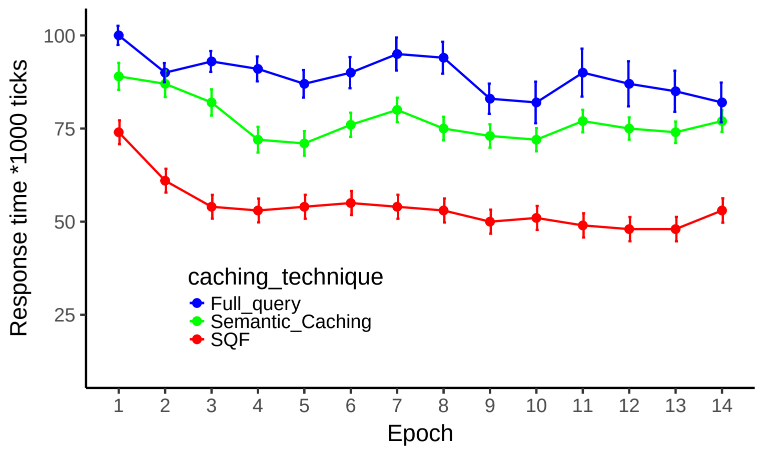


Figure 14: Response time due to inter-cache data transfers

Discussion:

The results in Figure 14 presented represent Uniform distribution of sub-queries. It is observed that the sub-query fragmentation (SQF) showed a clear advantage over other caching techniques used in distributed environment. SQF has fewer inter-cache transfers as this approach stores all related data segments together.

5.5 Overhead of Sub-query Fragmentation

Since sub-query fragmentation caches more frequently repeated fragments of a query, it is often possible to cache the same data segment as a part of several other sub-queries before it is identified as a frequent fragment. For example, a query Q has three sub plans : a, b and c. Assume, after some time it is found that a, b are as frequently queried together as a, c. The SQF decides to cache ac and ab as two sub-queries. A repetition of data for the fragment a. Where as, semantic segment caches a only once. So, SQF clearly tends to store more data initially. However, in the distributed environment, if the data is frequently needed at two locations at all times, then it makes sense to create copies of data and distribute them over different locations. This will have a significant reduction in the query response time. Since a is joined with b, the sub-query will be treated as an independent unit from ac in SQF. Here we could not present all our findings and methods to identify the association between frequently queried data segments due to lack of space.

6 Conclusion & Future Work

The work presented in this paper is a part of the research project - CommCache, a community shared cache framework for the optimization of data transfers. In this paper, we have discussed the problem of identifying suitable data fragments to cache in the distributed environment. The proposed sub-query fragmentation technique fragments user queries into sub-queries based on the repetition of partial query segments (sub-queries). They are stored together to provide quick retrieval of data. Though SQF is an extension to the semantic caching, sub-queries are modeled as portable objects suitable to the distributed environment.

Overall, the SQF approach performed better than other caching approaches in the distributed environment. For the average response time, SQF is very effective in dynamic environments where workloads are changing as it is able to adapt to those changes quicker and better than existing approaches. Using traditional workloads, SQF outperformed other methods in most of the instances. It is only second best, after SC for static workloads. Since SQF can be used find patterns of data accesses, it is possible to cache more accurate data than other techniques. Over a period of time, the percentage of data found is higher than other methods.

At present, SQF is restricted to the following limitations. The approach is mainly focused on distributed caches to understand the patterns of user queries using distributed learning. Hence a centralised global query processor is assumed to manage the distribution of query segments. In future, we would like to extend the use of SQF for de-centralised environments. Another limitation is, the SQF approach depends on query execution plans. This means, a different execution plan might lead to reprocessing of the entire query. However, the query planner can consult existing cached contents before creating a plan.

In future, we would like to extend the effectiveness of SQF in two directions: One, develop request-reply systems from approximated cached results. It is to rank sub-queries in the order of priority and formulate reply with existing cached contents. Second, several query fragmentation techniques and organisation of query execution are available for the execution of queries on shared memory and multi-core systems [29, 30]. It will be interesting to implement these techniques for distributed caches. We would like to pursue more study in this direction in future.

References

[1] Santhilata Kuppili Venkata, Jeroen Keppens, and Katarzyna Musial. Adaptive Caching Using Sub-query Fragmentation for Reduction in Data Transfers from Distributed Databases. In N. P. F. Lorente and K. Shortridge, editors, *ADASS XXV*, ASP Conf, Ser. ASP, 2016.

[2] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[3] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-server Database Architectures. *The VLDB Journal*, 5(1):035–047, January 1996.

[4] Qun Ren, Margaret H. Dunham, and Vijay Kumar. Semantic Caching and Query Processing. *IEEE Trans. Knowl. Data Eng.*, 15(1):192–210, 2003.

[5] Blesson Varghese, Nan Wang, Dimitrios S. Nikolopoulos, and Rajkumar Buyya. Feasibility of Fog Computing. *CoRR*, abs/1701.05451, 2017.

[6] Ivans Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *2014 Australasian Telecommunication Networks and Applications Conference (ATNAC)*, pages 117–122, Nov 2014.

[7] Mamta Agiwal, Abhishek Roy, and Navrati Saxena. Next Generation 5G Wireless Networks: A Comprehensive Survey. *IEEE Communications Surveys Tutorials*, 18(3):1617–1655, thirdquarter 2016.

[8] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A Study of Three Alternative Workstation Server Architectures for Object-oriented Database Systems. In *Proceedings of the 16th Intl Conf on VLDB*, pages 107–121. Morgan Kaufmann Publishers Inc., 1990.

[9] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems The Complete Book, 2 ed.* Pearson Prentice Hall, 2009.

[10] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–392. IEEE Computer Society, 2004.

[11] ChungMin Melvin Chen and Nicholas Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In Matthias Jarke, Janis Bubenko, and Keith Jeffery, editors, *Advances in Database Technology — EDBT '94*, pages 323–336, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[12] Björn Dór Jónsson, María Arinbjarnar, Bjarnsteinn Dórsson, Michael J. Franklin, and Divesh Srivastava. Performance and Overhead of Semantic Cache Management. *ACM Trans. Internet Technol.*, 6(3):302–331, August 2006.

[13] Norvald H. Ryeng, Jon Olav Hauglid, and Kjetil Nørvåg. Site-autonomous distributed semantic caching. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 1015–1021. ACM, 2011.

[14] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.*, 15(2):162–207, June 1990.

[15] Alfredo Goñi, Arantza Illarramendi, Eduardo Mena, and José Miguel Blanco. An Optimal Cache for a Federated Database System. *Journal of Intelligent Information Systems*, 9(2):125–155, 1997.

[16] Dongwon Lee and Wesley W. Chu. Towards Intelligent Semantic Caching for Web Sources. *J. Intell. Inf. Syst.*, 17(1):23–45, 2001.

[17] Dongwon Lee and W. W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proceedings of the Eighth International Conference on Information and Knowledge Management*, CIKM '99, pages 77–85, New York, NY, USA, 1999. ACM.

[18] Boris Chidlovskii and Uwe M. Borghoff. Semantic Caching of Web Queries. *The VLDB Journal*, 9(1):2–17, 2000.

[19] Jun Rao and Kenneth A. Ross. Reusing Invariants: A New Strategy for Correlated Queries. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 37–48, New York, NY, USA, 1998. ACM.

[20] Doaa Saad El Zanfaly, Reda A. Ammar, and Ahmed Sharaf Eldin. Modeling and analysis of a multilevel caching in distributed database systems. In *Proceedings of the 9th ISCC 2006, Egypt*, pages 140–145. IEEE Computer Society, 2004.

[21] Jason George McHugh. *Data Management and query processing for semistructured data*. PhD thesis, Stanford University, 2000.

[22] Laura M. Haas, Donald Kossmann, and Ioana Ursu. Loading a Cache with Query Results. In *Proceedings of the 25th International Conference on VLDB*, VLDB '99. Morgan Kaufmann Publishers Inc., 1999.

[23] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. *A Middleware System Which Intelligently Caches Query Results*, pages 24–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[24] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Active Proxy-G: Optimizing the Query Execution Process in the Grid. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–15, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[25] Beomseok Nam, Minho Shin, Henrique Andrade, and Alan Sussman. Multiple Query Scheduling for Distributed Semantic Caches. *J. Parallel Distrib. Comput.*, 70(5):598–611, May 2010.

[26] Laurent d'Orazio, Fabrice Jouanot, Yves Denneulin, Cyril Labbé, Claudia Roncancio, and Olivier Valentin. Distributed Semantic Caching in Grid Middleware. In *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, pages 162–171, 2007.

[27] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[28] Tamer M. Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[29] Rubao Lee, Minghong Zhou, and Huaming Liao. Request Window: An Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1219–1230. VLDB Endowment, 2007.

[30] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *Proc. VLDB Endow.*, 2(1):373–384, August 2009.