Contquer: An optimized distributed cooperative query caching architecture

by

Shamir Sultan Ali

School of Computer Science

McGill University, Montréal

January 2011

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

Abstract

The backend database system is often the performance bottleneck in multi-tier architectures. This is particularly true if there is a cluster of application servers while there is only a single database backend.

A common approach to scale the database component is query result caching. The idea is to cache the results of a query submitted to the database in a cache. If the query is consequently requested again, the result can be retrieved from the cache instead of the query again being submitted to the database. Query caching can play a vital role in reducing latency by avoiding access to the database, and improving throughput by avoiding a database bottleneck.

Existing approaches, however, have two limitations. First, they do not exploit the full capacity of the caches. Each application server has its own cache and frequently used objects will likely be cached in all caches, limiting the number of different objects and queries that can be cached. Furthermore, a query can only be served from the cache if previously the exact same query was posed.

In this thesis, we introduce Contquer, a distributed cooperative caching algorithm that uses a distributed caching architecture where each object is only cached at one application server and each application server has access to local and remote caches. Thus, the full capacity of all caches can be exploited. Furthermore, we optimize the query cache by exploiting the cache even if only part of a query can be served from the cache. For that we analyze the containment of queries within other queries. Contquer determines when a query can be fully or partially served from the cache, and

automatically generates remainder queries to the database if necessary.

This thesis reports on the design and implementation of Contquer. It also conducts experiments that show that performance is improved considerably with the proposed algorithm. We conclude that the use of a distributed caching infrastructure and the ability to retrieve partial results from the cache improves performance in terms of hit-rate, throughput and latency.

Résumé

Le système de base de données est souvent un point critique en terme de performance dans les architectures multi-tiers. Ceci est particulièrement vrai dans le cas d'un groupe de serveurs d'application alors qu'il y a seulement une seule base de données.

Une approche commune pour améliorer la performance de base de données est la mise en cache de résultat de requêtes. L'idée est de mettre en cache les résultats d'une requête soumise à la base de données. Si cette requête est demandée à nouveau, le résultat peut être récupéré à partir du cache au lieu de soumettre la requête à nouveau à la base de données. La mise en cache de requêtes peut jouer un rôle vital dans la réduction de latence en évitant l'accès à la base de données, et d'améliorer le débit en évitant la congestion de la base de données.

Les approches existantes ont cependant deux limitations. D'abord, ils n'exploitent pas la pleine capacité des caches. Chaque serveur d'application a son propre cache et des objets fréquemment utilisés seront probablement mis en cache dans tous les caches, ce qui limite le nombre d'objets et de requêtes qui peuvent être mis en cache. En outre, une requête ne peut être servie à partir du cache que si elle a déjà été servie de la base données.

Dans cette thèse, nous introduisons Contquer, un algorithme de mise en cache distribué et coopérative qui utilise une architecture de mise en cache distribuée où chaque objet est uniquement mis en cache à un seul serveur d'application et que chaque serveur d'application a accès à des caches locaux et distants. Ainsi, la capacité

totale de tous les caches peut être exploitée. En outre, nous optimisons le cache de requête en exploitant la mémoire cache, même si une partie seulement d'une requête peut être servie à partir du cache. Pour cela, nous analysons le confinement de requêtes dans les autres requêtes. Contquer détermine le moment où une requête peut être totalement ou partiellement servie à partir du cache, et s'il le faut génère automatiquement le reste des requêtes à la base de données.

Cette thèse porte sur la conception et la mise en œuvre de Contquer. Il mène également des expériences qui montrent que la performance est considérablement améliorée avec l'algorithme proposé. Nous concluons que l'utilisation d'une infrastructure de mise en cache distribuée et la possibilité de récupérer les résultats partiels de la mémoire cache améliore la performance en termes de taux de réussite, de débit et de latence.

Acknowledgements

I am thankful to my supervisor, Professor Bettina Kemme, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the problem. Her perpetual energy and enthusiasm in research had motivated me alot. In addition, she was always accessible and willing to help her students with their research. As a result, research life became smooth and rewarding for me.

I would like to thank all the members of the Distribution Information Systems Lab (DISL) team specially Kamal Zellag and Neeraj Tickoo for providing their support and help.

Moreover, I would also like to thank my parents, siblings and all my friends for always encouraging me and believing in me.

Table of Contents

List of Figures

Chapter 1

Introduction

The Internet has become a common model of interaction between businesses and customers in present time. Buying and selling products or services through the web has been popular for the last two decades and has seen revenue for many companies grow exponentially. It is thus very important that the customers are provided quick and easy service at all time. E-commerce is dynamic and constantly evolving, supported by technologies that are constantly changing. E-commerce web applications are generally based on a client/server model where the client is defined as the requestor of the service and a server is the provider of the service. To make this model scalable, e-commerce web applications are hosted in a multi-tier architecture, which is a software system segregated into separate layers.

A typical multi-tiered application consists of a presentation tier, a business tier and a data tier. The presentation tier is the presentation logic layer, something that the users view, but it does not expose anything about the internal structure of the system. A webpage being shown on a client's browser is such an example. The business tier, also often referred to as middle tier, is the brain of the entire application, as it provides business logic and controls data access. The business layer is usually

implemented within a web and/or application server. A web server usually delivers content such as web pages using HTTP. HTTP Requests from clients (usually through a browser) are forwarded to the application server causing the appropriate business logic to be executed. Based on the business rules set forth, the application server asks the data tier accordingly to get the required information. The data tier typically consists of a database system where information is stored and retrieved. The data retrieval process is most likely in the form of selecting, querying, inserting, updating, or deleting data stored at the data tier. The database is usually the bottleneck in a data-driven application. There are usually two approaches to scale the database component. One is to replicate the database; the other is to introduce layers of caching. Database replication is a complex process and is difficult to handle. In contrast, caching is a more common solution to avoid the database bottleneck.

A cache provides temporary storage of data. It improves performance by transparently storing data such that the future requests for the stored data can be served faster. A cache keeps a representation of recently accessed database content close to the application, either in memory or on disk of the application server machine. The cache is a local copy of the data. Caching could be incorporated at the business tier/middle tier before the database, or both. We look at middle-tier caching where the cache is located inside the application server.

Web applications often want to pose declarative queries (e.g., SQL queries) that return a set of objects that fulfill a certain predicate. The application sever can cache the results of SQL queries posed to the system into its memory and hence, improve the response time of subsequent queries to the same data by avoiding the need to make a database connection, reforming the query, re-executing it and retrieving the results. We stand to gain even more in terms of response times and resources used on systems where the database does not reside on the same machine as the application

server and requires a remote connection (TCP or similar). Also, during application processing, data is transferred from the database, transformed into business objects and manipulated by complex processes in the application server layer. If the ready-to use business objects are cached, this can dramatically improve the performance. Thus, caching queries and their results can dramatically improve response time and resource requirements.

Furthermore, performance of the overall system can be improved by scaling the business tier. This can be done through replication. Having multiple application servers, each having its own cache, results in reducing the number of queries that must be executed at the database system.

In this thesis, we introduce Contquer, a distributed cooperative query caching algorithm. Existing approaches for distributed query caching do not fully exploit the aggregate capacity of the caches. Having multiple caches without further coordination means that frequently used objects will likely be cached in all the caches, resulting in less objects cached overall. Therefore, we propose a cooperative caching architecture whereby every cache has knowledge of the contents of other caches. An object cached at a remote cache is transparently fetched from the remote cache as if the objet were cached locally. The full capacity of all the caches is exploited by having an object only cached at one application server.

Furthermore, query caching in most database systems can only serve queries if the exact same query was posed previously. To increase the query cache hit-rate and decrease access to the database, we optimize the query cache by using the cache content even if only part of the query result is cached. Contquer determines if a query can be fully or partially served from the cache, generating remainder queries to the database in case of partial hits.

1.1 Contribution

The thesis makes the following contribution:

- Improving query caching by not only serving exact queries from the cache but other forms of queries as well.
- Supporting query caching in a distributed environment by designing and implementing of distributed query caching algorithm – 'Contquer'.
- A detailed analysis of the impact of the distributed query caching framework.

1.2 Thesis Outline

This thesis is divided into 5 chapters (including this introduction chapter). Chapter 2 provides background on the architecture of typical web applications, and a general overview of how query caching works in such environment. It also explores the related work done in the context of this thesis, which helped us to form the base of our research, and the ways in which our approach differs from them. In Chapter 3 we present and discuss in detail our Contquer algorithm. Chapter 4 discusses the distributed cooperative caching architecture and how Contquer works in such distributed environment. Chapter 5 reports performance evaluations for the model proposed. Finally, Chapter 6 presents our conclusions and outlines some possible future research work in this domain.

Chapter 2

Background and Related Work

2.1 Introduction

In the following section we first describe a typical web application architecture. We then also describe the general structure of queries with the help of some example queries and show how the simple query cache works. Then we discuss how the caching ideas are integrated into a web application architecture. Related work in the area of query caching is also discussed.

2.2 Typical Web Application Architecture

Figure 1 shows a typical architecture for web sites serving dynamic content. The web server receives the request (HTTP) from a client (usually through a browser), which is then forwarded to the application server causing the appropriate application logic to be executed. The application in turn sends queries, depending upon the client request and application logic, to the database. Upon receiving a reply from the database for the queries, the application server then constructs a reply, which is then sent through

the web server back to the client.

In a three-tier architecture, as the one discussed here, web server and application server constitute the middle-tier. A web server usually delivers content such as web pages using HTTP. Some examples of web server technology include Apache Tomcat, IIS from Microsoft, OpenLink etc. The role of the application server is to provide business logic for an application program and also middleware services for security and state maintenance, along with data access. Glassfish, JBoss, and WebLogic are a few examples of widely used application servers. Usually application servers have a built-in web server component for receiving requests and returning responses to the client. For the purpose of this thesis, we consider the combination of web server and the application server collectively as application server.

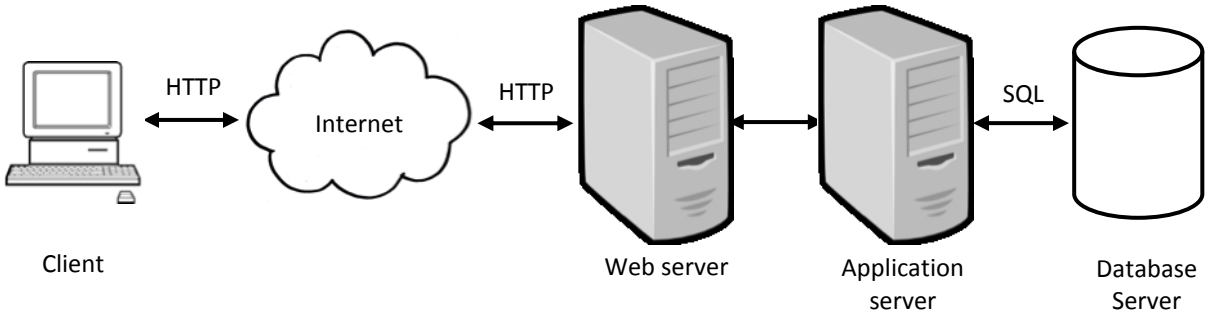


Figure 1: Common architectures for web-sites

2.3 SQL Queries and Query Parameters

In this section we introduce some example queries and terms related to query caching that will be used in the following sections. SQL, often referred to as Structured Query Language, is a database language primarily designed for managing data in RDBMS (relational database management systems).

A query consists of several clauses as shown in Figure 2.

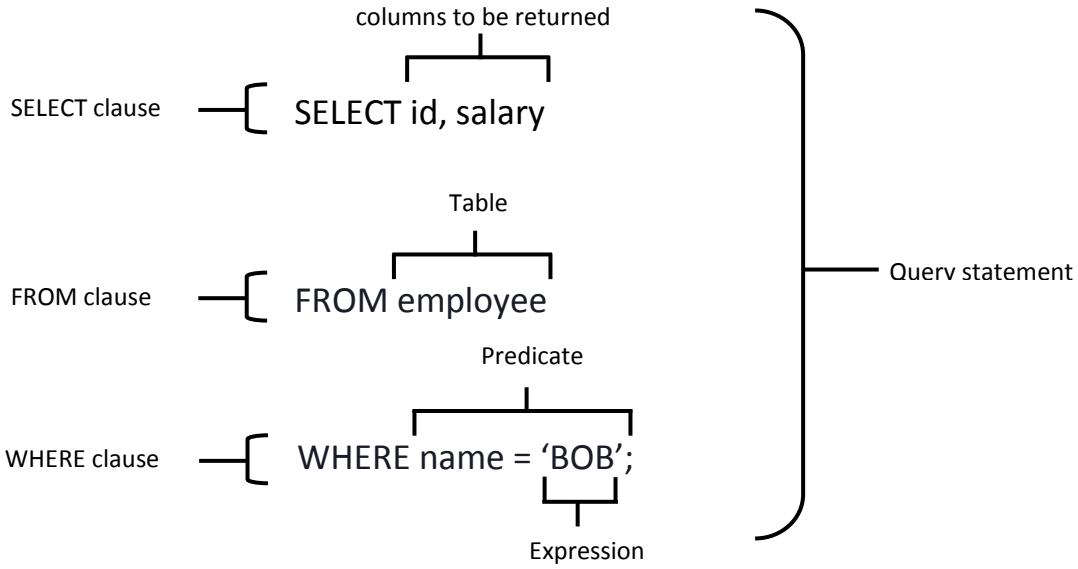


Figure 2: Elements of a simple SELECT query statement

The most common clauses used in a query are:

- The SELECT clause indicates the attributes (columns of a table) to be retrieved. For example in the query below, only the name and age attributes from the employee table are returned.

  SELECT name, age
  FROM employee

  If all attributes should be returned, SELECT * is used as shown by the query below.

  *SELECT **
  FROM employee

- The FROM clause indicates table(s) from which data is to be retrieved. The FROM clause can also include optional JOIN sub clauses to specify the rules for joining tables.
- The WHERE clause is a Boolean expression, with operators AND, OR and NOT, where each literal is a comparison predicate. Each comparison predicate evaluates to either FALSE or TRUE, determining the Truth-value of the entire Boolean expression.

The comparison predicates can be sub-categorized as "equality" and "inequality" comparison predicates:

- Equality predicates have a comparison based on equality (= equal sign). The =

(equals) comparison operator compares two values for equality. Most of the equality predicates compare the contents of a table column to a value, as in the example query below. A comparison expression may also compare two columns to each other.

*SELECT **
FROM employee
WHERE name = 'BOB';

This query returns all employee records that have in their column *name* the string value 'BOB'.

- Comparison operators other than equal lead to inequality predicates. Examples are ">" (greater than), ">=" (greater than or equals to), "<" (less than), "<=" (less or equals to), "<>" (not equal to). An example for inequality predicate is shown in the example query below:

  *SELECT **
  FROM employee
  WHERE salary > 50000

  This query returns all employee records that have a salary attribute value above 50000.

Complex queries combine several comparison predicates in a Boolean expression using the AND, OR and NOT operators. Some examples are:

Query I:

*SELECT **

FROM employee

WHERE city = 'Montreal' AND salary > 50000 AND salary < 75000;

Query I has both equality and inequality predicates in a Boolean AND-expression. The query returns all records of the employee table where the city attribute has the value 'Montreal' and the salary attribute is between 50000 to 75000.

Any arbitrary combination of AND, OR and NOT operators can be used. In many cases, a NOT operator transforms an inequality predicate into another inequality predicate. For instance, the two following predicates (P1 and P2) are equal.

P1: NOT salary < 5000

P2: salary >= 5000

Therefore, we ignore the NOT operator for inequality predicates in the thesis.

In query II, all records of the employee table are returned where the city attribute either has a value of 'Montreal' or 'Toronto', and the salary attribute is greater than 50000.

Query II:

SELECT name, position, salary

FROM EMPLOYEE

WHERE (city = 'Montreal' OR city = 'Toronto') AND salary > 50000;

There could be multiple conjugations within a single query as shown in Query III:

Query III:

SELECT name, position, salary

FROM EMPLOYEE

WHERE (city = 'Montreal' OR city = 'Toronto') AND

(department = 'Marketing' OR department = 'IT') AND

salary > 50000;

Any Boolean expression can be transformed into a disjunctive normal form (DNF). A DNF is a disjunction (sequence of ORs) consisting of one or more disjuncts, each of which is a conjunction (AND) of one or more literals [17]. The only operators in DNF are AND, OR and NOT. The NOT operator can only be used as part of the literal, preceding a variable. In DNF, the query can be processed as one or more independent conjunctive queries (having AND operators only) linked by UNIONS (sequence of ORs). For instance Query III can be re-written into disjunctive normal form as:

SELECT name, position, salary

FROM EMPLOYEE

WHERE (city = 'Montreal') AND (department = 'Marketing') AND salary > 50000

OR

(city = 'Montreal') AND (department = 'IT') AND salary > 50000

OR

(city = 'Toronto') AND (department = 'Marketing') AND salary > 50000

OR

(city = 'Toronto') AND (department = 'IT') AND salary > 50000 ;

In general, a DNF is represented as:

$$(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee (p_{21} \wedge p_{22} \wedge \cdots \wedge p_{2n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$$

where each p_{ij} is an equality predicate, possibly preceded by a NOT operator, or an inequality predicate.

In this thesis, we distinguish two different query types. Query type 1 consists of comparison predicates that are connected through AND operators only. Query type 2 has OR operators in addition to AND operators. Thus, the query needs to be transformed into a DNF. In general, we do not consider NOT operators.

2.4 Scope and Limitation

In this thesis, we only consider queries involving one table. Joins and other multi-table queries are difficult to handle in dynamic environments and thus, not considered here. Also, only the most widely used operators and expressions are supported. Inequality operators such as greater than (>), greater or equal (>=), less than (<), less or equal (<=) are supported. Other relational operators that are supported are equals (=), BETWEEN and IN. The NOT IN and LIKE operators are not supported.

2.5 Flow of a simple Query Cache

Database access is expensive as it consists of many steps. Avoiding such database access is necessary since it is a time consuming process. It includes:

- Connection to the database
- Preparation of the SQL query
- Sending the query to database
- Formation of the query plan at database
- Execution of the query
- Retrieving the results
- Closing the database connection

The above process is quite resource intensive and can adversely affect systems performance. Also, during application processing data is transferred from the database, transformed into business objects and manipulated by complex processes in the application server layer. Having multiple parallel connections at the database and fetching huge result sets from the database usually results in the database becoming the bottleneck. Thus, a query cache can result in improving performance of the overall system and avoiding bottleneck.

The query cache can be useful in an environment where we have tables that do not change very often and for which the server receives many identical queries. This is a typical situation for many web servers that generate many dynamic pages based on database content. For example, an e-commerce website may have many web pages

listing all the products for a specific category that they are selling (e.g. books, electronic items, services, etc). By caching content generated by executing one or more queries at the database, we can greatly increase the response time by fetching the content from the cache rather than going to the database. Thus, we reduce the number of queries that must be executed at the database system and also save the execution time.

Figure 3 shows the typical flow of query execution as currently found in most application servers that offer a simple query cache.

Query execution is triggered upon receiving a query from the application. First, the query is looked up in the query cache. If there is an exact match (the exact same query with the same attributes in the select clause and the same Boolean expressions in the where clause exists in the cache), the relevant tables involved in the query are checked. If any of the tables was changed since the query was cached, the query has become stale. If the query is not stale, the cached results are returned to the client. Otherwise, if the query is found to be stale it is removed from the cache, the query is executed again, and the results are cached before returning them to the client.

Whenever an UPDATE, INSERT or DELETE statement is submitted, some of the cached queries might become stale. A simple and common approach uses a table level invalidation. We assume such an approach in this thesis. It is achieved by having timestamps assigned to every table and every query. The timestamp of a table increases whenever the table changes. A query receives the current timestamp of the table referenced in the query.

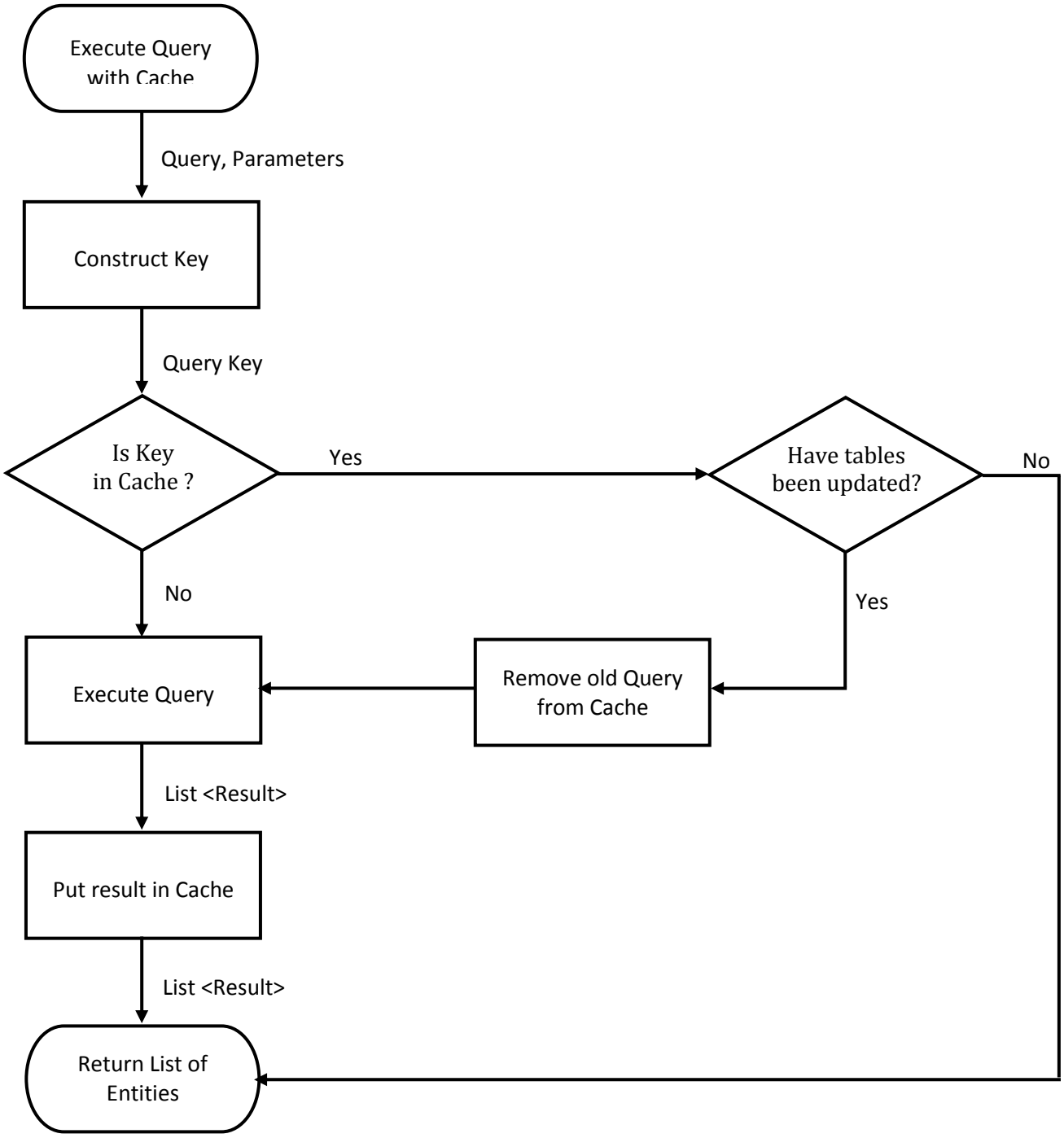


Figure 3: Query execution with simple query cache

Thus, upon finding a candidate query (CQ) in the cache, we compare the current timestamp of the table referenced in the query with the timestamp of the candidate query. There are two cases:

1. Timestamp of the table = timestamp of the cached query (CQ); this means that the query was cached after the table was last modified, and thus, the cached query result is not stale and can be used.

2. Timestamp of the table > timestamp of the cached query (CQ); this means that the table was modified after the query was cached. Therefore, the cached query is invalidated and removed from the cache. The query is then sent to and executed at the database. The query and its results are cached and the current timestamp is assigned to the newly cached query.

In case where the incoming query is not cached, the query is immediately sent to the database, executed and results are stored in the cache before returning them to the client. If the same query is called again, the query results can be fetched from the cache and the query does not need to be executed again.

2.6 Full and Partial Containment

If only exact query matching is done, the hit ratio can be small. The question is whether we can take advantage of cached queries even if they only partially help the currently posed queries. For that let's have a look at containment. Full and partial containment happens when a query response is completely or partially contained within one or more cached query result sets.

2.6.1 Full Containment

Full containment occurs when the query response can be satisfied fully from the cached query itself and there is no need to consult the database for returning the query response to the client. There could be different scenarios that could result in full containment.

2.6.1.1 Exact Query Matching

We discussed exact matching before where the cache contains exactly the query that is posed. For that, the query string must be identical. For example, in Figure 4, the incoming query Q is identical to the cached query C.

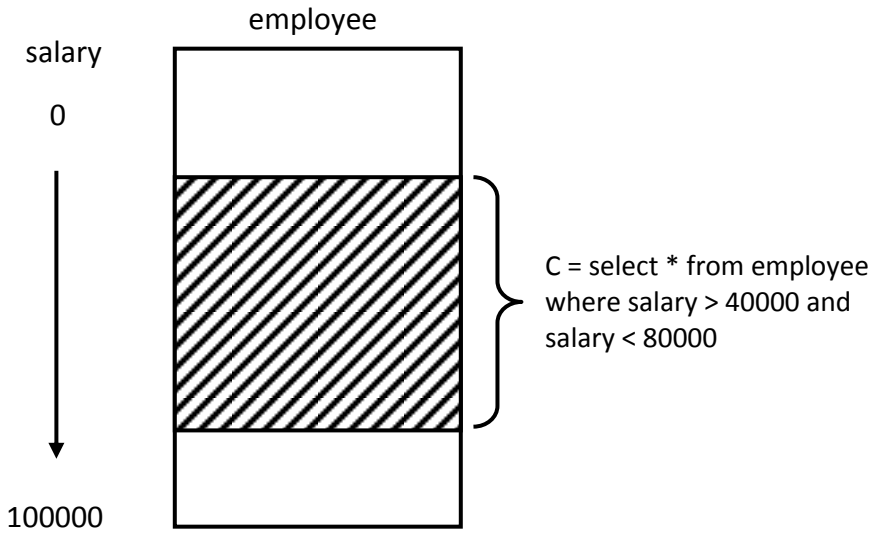


Figure 4: Full containment of query response with an exact match
*Q = select * from employee where salary > 40000 and salary < 80000*

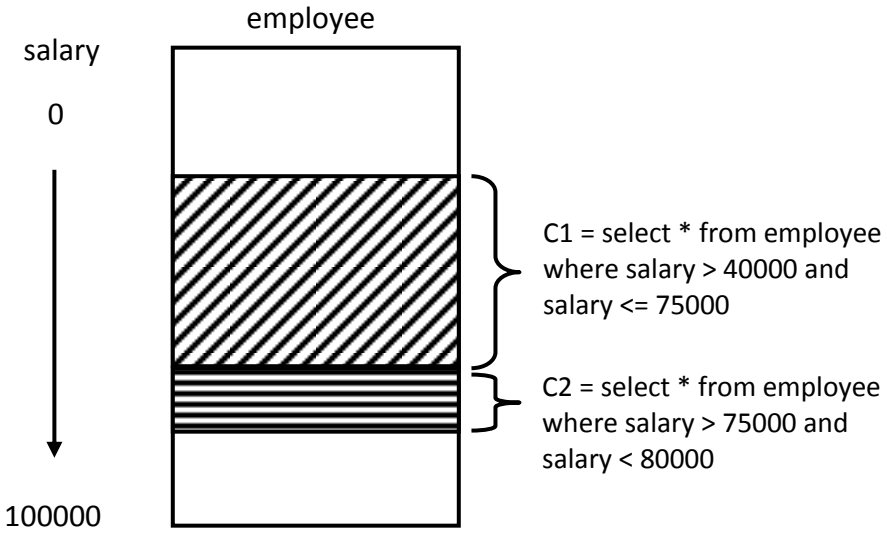


Figure 5: Full containment of query response with multiple subset queries
*Q = select * from employee where salary > 40000 and salary < 80000*

2.6.1.2 Multiple Subset Queries

Multiple cached queries can also result in full containment. More than one query, each having as result a subset of the result of the incoming query, can be combined to get the required response. For example, in Figure 5, merging the result sets of two already cached queries (C1 and C2) satisfies response of the incoming query Q. Here, the multiple cached queries might have overlapping result sets. Therefore, the duplicates must be filtered out.

2.6.1.3 Superset Query

Full containment is also given if there exists a cached query whose result set is a superset of the current query's result set. In other words, there are more tuples cached then the required response. The situation is depicted in Figure 6. In this case,

however, there needs to be a mechanism whereby the extra tuples are removed from the cached result set before the response is returned to the client. This can be difficult. For instance, information required for this filtering mechanism is missing if the attributes involved in the comparison predicates of the where clause are absent in the select clause.

For example, the current query (Q) requests for all the tuples in the employee table where the salary attribute value is more than 60000 and less than 75000:

Q: *SELECT name, address, city*
FROM EMPLOYEE
WHERE salary > 60000 AND salary < 75000;

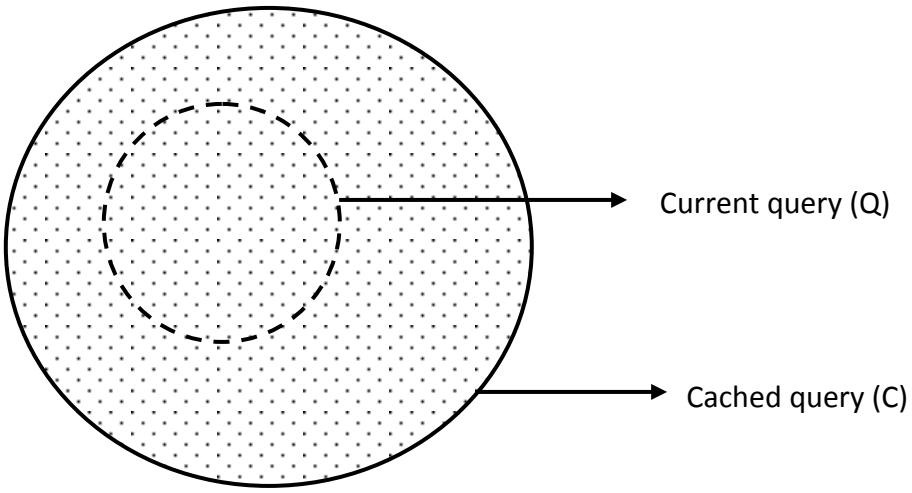


Figure 6: Full containment of a query response with a superset query
The cached query is a superset of the current query response.

The following query C is a potential candidate for a full containment for Q as its result set has all employee records where the salary attribute value is more than 50000 and less than 100000.

C: *SELECT name, address, city*
FROM EMPLOYEE
WHERE salary > 50000 AND salary < 100000;

Both the above queries (Q and C) have only three columns (name, address and city) in their result set. However, in order to find the subset of tuples in the result set of C that fulfill the where clause of Q, we would need the salary column. Therefore, the filtering mechanism cannot proceed and the cached results cannot be used. Hence, it is necessary to have all the required information in the result set of the cached query (SELECT clause) to filter out such extra tuples. Even if all the information is present, the cache has to implement the necessary filter mechanism, thus, partially copying the functionality of the database. Although in our infrastructure we do have all the required information we do not consider superset queries in this thesis, as it requires extra query processing which is difficult to achieve.

2.6.2 Partial Containment

While full containment can return the entire result form the cache, partial containment has only a subset of the necessary tuples in the cache. The additional records, which are not yet cached, are fetched from the database. Similar to full containment, it is possible that multiple cached queries can be used to get as much partial result as possible.

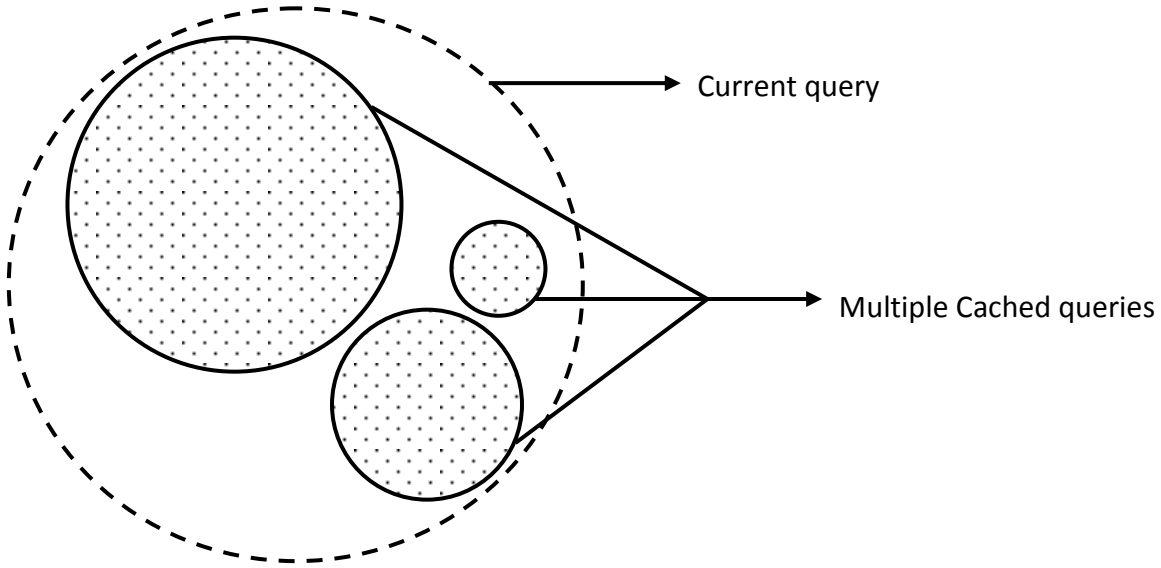


Figure 7: Partial containment of a query response
Multiple cached queries build a subset of the current query response.

The missing tuples that are not yet cached, can be obtained by generating a remainder query that is then sent to the database. In the example of Figure 7, three partially contained cached queries are used to get a partial response for the current query. The remaining tuples (represented by white space) are fetched from the database by generating one or more remainder queries.

All cached result sets and the records from the database are then merged and the final merged query result is returned back to the client. For example, in Figure 8, C1 and C2 can partially respond to the incoming query Q. The missing tuples with salary attribute values between 60000 and 75000 are fetched from the database system by generating a remainder query. The tuples from the cached results of C1 and C2, and from the database are merged and returned back to the client.

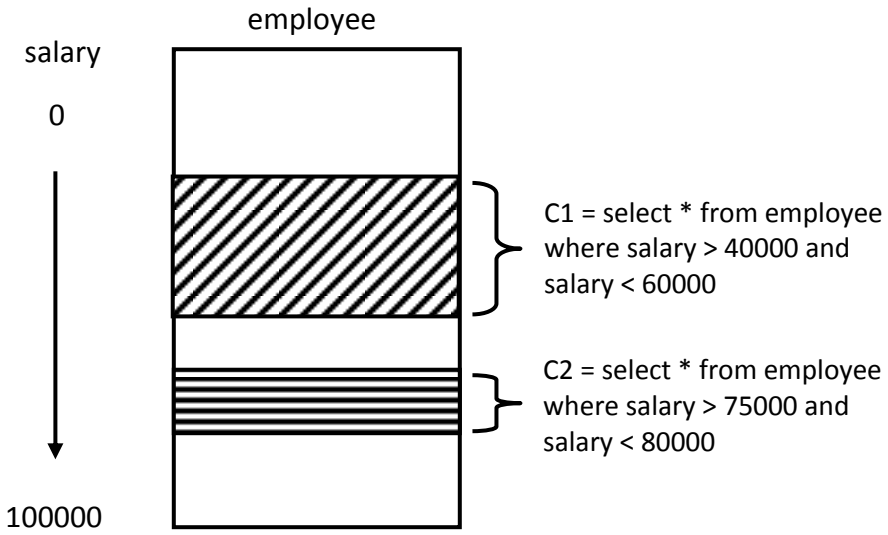


Figure 8: Partial Containment of a query response with two cached queries
*Q = select * from employee where salary > 40000 and salary < 80000*

2.7 Caching in JBoss Application Server

The cache in a 3-tier architecture is usually located on the middle tier as part of the application server enivronment. As we discussed in the introduction chapter, one of the ways of scaling the overall system is to replicate the business tier. This means having multiple application servers each of them having their own cache. Therefore, it is important to understand how caching, and in particular query caching, works in a distributed environment.

To better understand how the cache and the query cache work we should know how application servers interact with their cache. The JBoss Application Server (JBoss AS) is a widely used open-source Java EE based application server. We have used the JBoss

AS to carry out all our experiments. The JBoss AS provides many features, among them the Hibernate object-relation mapping. Hibernate acts as a mapping tool from Java classes to database tables, thus replacing direct persistence related database accesses with high-level object handling. In general each row of the database is represented as an object in Hibernate. Hibernate also provides data query and retrieval facilities, and uses a two-level cache architecture as seen in Figure 9.

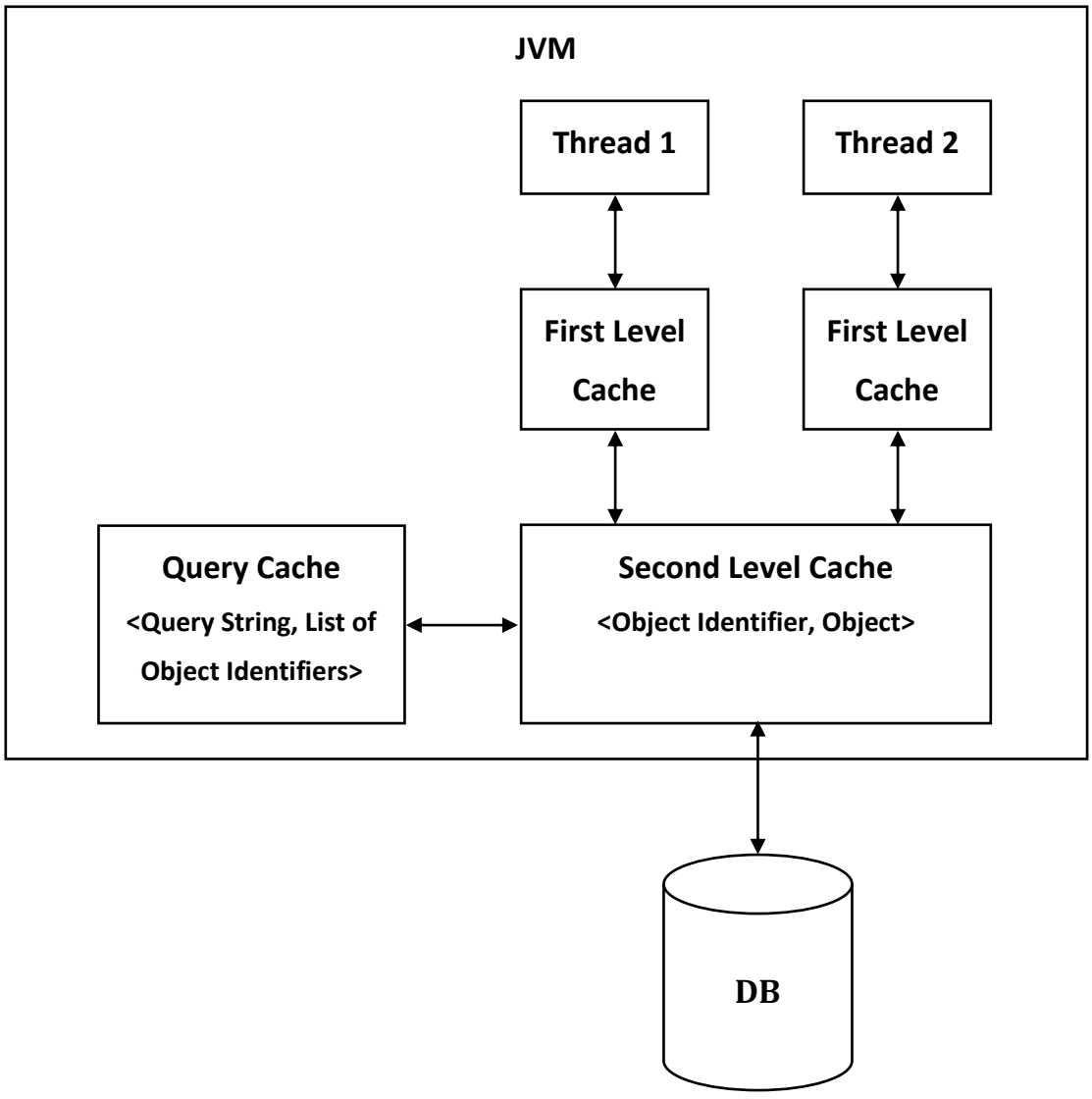


Figure 9: Hibernate Caching Architecture

- The first-level cache in Hibernate is a local session level cache. A page request from a client may require a transaction to be executed on the database. This transaction might contain different queries that need to be executed. The first level cache here only caches objects and queries pertinent to that particular transaction. When the transaction completes the cache is no longer available and is destroyed. Thus, multiple client requests each have their own first level cache. This can also be seen in Figure 9. Thread 1 and thread 2 represent different requests from either the same client or different clients, having their own first level cache.

- The second-level cache in Hibernate is pluggable and has as scope the whole application across many client requests. For example, a website XYZ hosted by an application server gets web page requests from various clients. For all the requests, or in other words, for the whole application, there is only one global cache, which is second level cache. Thus, all client requests have access to this second level cache. The second-level cache has object identifiers that are mapped to the object themselves. We have used EHCache [10] as the second-level cache.

- Hibernate also implements a query cache that is integrated closely with the second-level cache as seen in Figure 9. It works as discussed in Section 2.5. It caches query strings and a list of object identifiers that are in the queries' result sets. Those object identifiers are then mapped to their respective objects in the second-level cache. A query is removed/evicted from the query cache either by invalidation or because of memory constraints. In this case, only the query string and corresponding object identifiers of the query's result set are removed from the query cache and all respective objects remian in the second-

level cache. Similarly, if an object is removed/evicted from the second-level cache the queries that contain that objects identifier in their list remain in the query cache. If a client makes a request resulting into one of those queries, the query still exists in the query cache and also its corresponding object identifiers in its result set. But the references of objects that were evicted no longer point to the second-level cache. Instead, these objects are fetched from the database and put into the second-level cache.

When a query is executed for the first time (it is not in the cache) the query is rewritten to only retrieve the object identifiers. Then, in a second step, only the objects that do not yet exist in the second-level cache are retrieved and put in the second-level cache.

It is possible for a query, which does not already exist in the query cache but some of the objects in its result set might already be present in the second-level cache. In that case only the objects that do not exist in the second-level cache are fetched from the database and put into second-level cache.

When the client requests a page, typically through a browser's HTTP request, it is received by the web server and is then forwarded to the application server. Here, based on the page requested, the business logic is executed which usually results in forming a SQL query. As we discussed, there are two levels of cache in Hibernate. Upon formulating a SQL query at the application server the query is looked up in the first level cache and then, if not found, in the second level cache.

Second level cache is something that is interesting to us as it represents a shared cache of the whole web application. For example, client X requests all products priced less than $2000. Lets denote this with Q1. If it is neither found in the first level cache

nor in the second level cache, it is submitted to the database and the resulting objects are cached in both the first level cache and the second level cache. If another client Y requests the same list of products, the first level cache will not have the query cached as a different client has requested it. But the second level cache, which is global for all the client requests, has the resulting objects, and the results are returned from the second level cache rather than fetching it from the database again.

2.7.1 Memory Costs for Query Cache

As discussed that queries cached in query cache only store query result identifiers and corresponding objects are cached in second-level cache. Thus, memory costs for query cache is minimal as compared to second level cache because query cache only caches query string and object identifiers in its result set where as second level cache contains all the objects.

2.8 Related Work

Database query caching has been proposed by many to improve the throughput of the whole system [1, 5, 6, 7, 8]. The idea is to cache the database queries and their results in order to improve latency and reduce the load on the back-end database server(s). If the database server is the bottleneck, query caching has the potential to greatly improve performance of the overall system.

Caching is an integral part of object-oriented database systems. Distribution of such a cache is considered by [1]. The paper analyzes the effect of query caching on a typical three-tier architecture maintained with multiple application servers. The paper proposes full and partial containment of query results to reduce the number of cache misses. This is primarily achieved by use of templates. For every query possibly submitted, a template has to be provided in advance. We illustrate this by the following example taken from the paper:

T: (SUBJECT = ?) AND (PRICE < ?)
Q: (SUBJECT = ‘KIDS’) AND (PRICE < 100)
C1: (SUBJECT = ‘KIDS’) AND (PRICE < 50)
C2: (SUBJECT = ‘ARTS’) AND (PRICE < 100)

The template T describes the WHERE clause of a query indicating all the predicate comparisons but not the actual values to be compared to. That is, a template represents a parameterized query. Only AND operators are considered in the paper. A posed query Q or a cached query C provides concrete values for the parameters. For an incoming query Q of template T, all cached queries C_i of template T are checked for partial containment. In above example, we can see that C1 is partially contained in Q but C2 is not. Thus, C1’s response along with the response from a remainder query

suffices to get Q's response. The design is pretty simple but is conservative as it is based on templates that have to be defined in advance.

The paper also discusses the effect of invalidations as a result of writes by evaluating coarse-grain table-level invalidation versus fine grain column-level invalidations. Table level invalidation, as discussed in Section 2.5, invalidates all queries on a table whenever any change on the table occurs. This may cause many queries to get invalidated even if the update to the table may not have altered the invalidated query's result set at all. In contrast, with column level invalidation only the queries that contain a column that was changed by the update, are invalidated. [1] shows that this greatly reduces the number of incorrect invalidations.

The paper also evaluates different system designs such as a purely distributed cache as well as a central cache in front of the database. The closer the cache is to the front-end, the shorter will be the latency to get the data from the cache. Therefore, the paper looks into tradeoffs of having the cache at different locations within the system. The paper, however, does not explore distributed query execution over several caches and how the different caches can cooperate with each other.

The location of the query cache, e.g. whether it is part of the server system or residing in a proxy cache close to the user, has been explored by [8, 12].

Cooperative caching has been considered by [11, 13, 14, 15] in peer-to-peer systems with the idea to retrieve data from the caches of nearby nodes instead of from the database system. However, the solutions are quite different to what is required to be done in a cluster environment with dynamic data.

In the context of web-caching, cache sharing has been explored in several ways. For

instance, in order to find web pages in neighboring caches, distributed [3] or centrally managed [4] dictionary information could be used. The interrelationship between caches or the distribution of caches has not received a lot of attention.

2.8.1 Query Caching in Database Systems

MySQL Query Cache caches queries and their results at the database, just as described in Section 2.6.1.1. The query cache stores the text of a SELECT statement together with the corresponding result that was sent to the client. Upon receiving an identical query later, the server retrieves the results from the query cache rather than parsing and executing the statement again [9]. This eliminates executing a query again but only if there is an exact match. Other commercial databases just cache the query plan and not the result, therefore, eliminating the need for making a query plan whenever the same query is executed. However, the query itself has to be executed again.

Chapter 3

Contquer: An Optimized Query Caching Framework

3.1 Introduction

Supporting declarative queries has a huge potential for reducing the load on the database. However, as discussed in the previous chapter, the simple query-caching model only matches queries that are exactly the same. Thus, in this thesis, we develop a query caching framework that can serve queries from a cache whenever full and partial containment is given. The algorithm developed in this thesis, called *Contquer*, looks for all disjoint and partially contained queries that can fully or partially satisfy a current query response.

An existing open-source cache (EHCache) is extended to provide support for such sophisticated query caching. EHCache is an application server based cache used to boost performance, offload the database and simplify scalability [10]. Like most other caches it only supports the simple query cache model.

3.2 Contquer Algorithm

To better understand the Contquer algorithm, the following example queries are used.

QUERY A:

*SELECT **

FROM employee

WHERE name = 'BOB' AND city ='Montreal';

QUERY B:

*SELECT **

FROM EMPLOYEE

WHERE city = 'Montreal' AND salary > 50000;

QUERY C:

*SELECT **

FROM EMPLOYEE

WHERE city = 'Montreal' AND salary > 40000 AND salary < 80000 AND age > 25;

QUERY D:

*SELECT **

FROM EMPLOYEE

WHERE (city = 'Montreal' OR city = 'Toronto') AND salary > 50000;

In the following, we first present a high-level algorithm of the Contquer algorithm that has been developed in this thesis.

Input:

Q: Incoming Query

All we need is just the incoming query, and the Contquer algorithm undertakes the following five steps.

Step 1: Query transformation

Any incoming query is transformed into a disjunctive normal form (DNF) first. Query A, B and C are already in DNF, but Query D is not, and therefore, is transformed into DNF as:

*SELECT **
FROM EMPLOYEE
WHERE city = 'Montreal' AND salary > 50000
OR
city = 'Toronto' AND salary > 50000;

Once a query is in DNF, OR-operators are always outer operators. In a next step, whenever a query contains OR-operators, it is split into sub-queries such as none of the sub-queries contains an OR-operator. In the above example, only D has to be split, separating it by the OR-operator gives two queries:

Query 1:

```
SELECT *
FROM EMPLOYEE
WHERE city = 'Montreal' AND
      salary > 50000;
```

Query 2:

```
SELECT *
FROM EMPLOYEE
WHERE city = 'Toronto' AND
      salary > 50000;
```

Query 1 and Query 2 are considered as two separate and distinct queries. The Contquer algorithm takes each of the query to see whether it can be partially or fully served by the cache. Once the results have been calculated from the cache and/or the database for each of the two queries, the result sets are merged and duplicates are eliminated. Thus, the Contquer algorithm itself only handles queries that contain AND-operators. Furthermore only equality predicates can be preceded by a NOT-operator.

Step 2: Identifying initial candidate set

This step identifies all queries already cached that are similar to the incoming query and are potential candidates. It results in:

C: List of cached queries similar to Q that are potential candidate queries for full or partial containment. A candidate query must be over the same table as the incoming query and have predicates over the same attributes.

Details for identifying initial candidate set are discussed in Section 3.3.

Step 3: Exact match lookup

In this step the Contquer algorithm checks whether there is an exact match of Q in the query cache. If so, the query result already cached is returned to the client. Exact match means that all the equality and inequality predicates must exactly match a cached query. For example assume step 2 gives initial candidate set C = {C1, C2 ,C3} for the queries shown below, where C2 is an exact match to Query A. Thus C2's result set is sent back to the client.

Query A: *name = 'BOB' AND city ='Montreal';*

C1: *name = 'TOM' AND city ='Montreal';*

C2: *name = 'BOB' AND city ='Montreal';*

C3: *name = 'TOM' AND city ='Toronto';*

Step 4: Candidate set refinement

If no exact match is found, the next steps 4a and 4b look for containment within the initial candidate set. Queries that do not qualify for containment are removed from further consideration. That is, the candidate set is refined at each step.

Step 4a: Check equality constraints

One condition is that the incoming query and the candidate queries must have the same equality predicates. Therefore, all queries are removed from the candidate set that do not have the same equality predicates as the incoming query.

FOR EACH C_i in C

IF equality predicates are not the same as in Q THEN

remove C_i from C

END IF

END

For example, for Query B, assume there exist candidate set C = {C4, C5, C6, C7, C8}. C5 is eliminated since it does not have the same equality predicate as Query B. The candidate set is updated to C = {C4, C6, C7, C8}

Query B: *city = 'Montreal' AND salary > 50000;*

C4: *city = 'Montreal' AND salary > 60000;*

C5: *city = 'Toronto' AND salary > 75000;*

C6: *city = 'Montreal' AND salary > 40000;*

C7: *city = 'Montreal' AND salary > 75000;*

C8: *city = 'Montreal' AND salary > 50000 AND salary < 55000;*

Step 4b: Check inequality constraints

The next step compares inequality predicates. For every query C_i in the updated candidate set C, the range of each inequality predicate is compared to that of Q. If for any inequality predicate, the range of that predicate of C_i is not a subset of or equal to the range of the corresponding predicate of Q, C_i is removed from the candidate set C. That is, to remain a candidate query, C_i's ranges in the inequality predicates must be subsets of the ranges of the corresponding predicates in Q.

FOR EACH C_i in C

FOR EACH inequality comparison predicate in C_i

IF range of predicate in C_i is not a subset of corresponding predicate in Q THEN

remove C_i from C ;

END IF

END

END

Continuing with the previous example of Query B, we had so far C = {C4, C6, C7, C8} as candidate set. Considering one cached query at a time, the range for each inequality predicate, which in this case is only the salary attribute, is compared to the range for the same inequality predicate of Q. Inequality predicate ranges for C4 (salary attribute above 60000), C7 (salary attribute above 75000) and C8 (salary attribute between 50000 and 55000), are a subset of that of Q (salary attribute above 50000). Therefore C4, C7 and C8 remain in C. The predicate range of C6 (salary attribute above 40000), however, is a superset of Q's range. Therefore it is eliminated from the candidate set. As a result, C is now {C4, C7, C8}.

Step 5: Candidate selection and Iteration

If the candidate set C is now empty, then no containment queries are found in the cache, and Q is sent to the database. Otherwise, if there exist queries in C, the query C_i with the largest result set is chosen. Note that we look at the size of the result set and not the size of the range because we want to choose the query that minimizes the number of tuples that must be retrieved from the database.

The missing tuples, i.e. those that are in the result set of Q but not in that of selected candidate query C_i, have to be retrieved. This can be done by generating a remainder query $R = Q - C_i$ that returns exactly those tuples.

IF C is empty THEN

send Q to database

ELSE

choose $C_i \in C$ such that C_i has largest result set of all queries in C ;

Prepare a remainder query: $R = Q - C_i$;

remove C_i from C ;

Continue from Step 4b ;

END IF

R could be sent to the database. But it might also be possible that a query is cached, that is an exact match for R or that is a containment query for R. Therefore, the Contquer algorithm is reiterated, R now becomes the incoming query Q, and the algorithm is continued from Step 4b. As the inequality predicates are only changed in the remainder query, the candidate set thus far refined can be used and comparison of inequality predicates takes place as in Step 4b. Such iteration is done until R has no containment within C, and thus, R is sent to the database.

Continuing with the example of Query B, and C = {C4, C7, C8}, the cached query with the largest number of tuples is selected as a partial containment query. Assume this is C4. C is updated to C = {C7, C8}. A remainder query R1 is generated to get the missing tuples, i.e., those that must be returned by Q but that are not returned by C4.

R1: *city = 'Montreal' AND salary > 50000 AND salary <= 60000;*

R has all the equality predicates of Q; only the inequality predicates are changed so as to get the missing tuples. R now becomes the new incoming query Q, and the algorithm continues from Step 4b. C8 is found as a partial containment query for Q in the candidate set C, and a new remainder query R2 is generated as:

R2: *city = 'Montreal' AND salary >= 55000 AND salary <= 60000;*

This R2 becomes the next incoming query Q and the containment algorithm continues from Step 4b comparing the inequality predicates of Q to the queries in candidate set C, which right now is C = {C7}. C7 does not qualify for the partial containment and upon finding no partial containment for R2, it is sent to the database.

Step 6: Merge

In the last step all the result sets from the cached queries that were selected by the Contquer algorithm, and the result set from the remainder query sent to the database, are merged and returned to the client.

In our example, the response for Query B is generated by merging the result sets from the cached queries C4 and C8, and from the query R2 that was sent to the database.

Some aspects of the Contquer algorithm are discussed in the next few sections in detail.

3.3 Query Transformation and Identifying Initial Candidate Set

The first and an important part of the Contquer algorithm is to identify all queries in the cache that are similar to the incoming query (Q) and can be considered as potential candidate queries for containment of Q's response. This could simply be achieved by using templates as done by [1]. But for that, all query types of the application have to be known in advance. This requires upfront intervention and is application dependent. This means that the approach is only conservative and can only match queries for which the templates have been provided.

Instead, in this thesis, we opted to match queries dynamically, and no application dependent generation is needed. Every incoming query is first parsed and a query tree is built. An example query tree is shown in the Figure 10. The query tree helps in identifying if the incoming query is in DNF or not. If not, the query is transformed into a DNF and split into several queries based on the number of disjunctions of conjunctive clauses in the WHERE clause.

For example, the query below (only where clause shown here), which produced the query tree of Figure 10, is first converted into DNF, then is further split into three different queries that are considered as three distinct incoming queries for the algorithm:

Q: *salary > 50000 AND city = "Montreal" AND ID < 100*

OR

salary > 50000 AND city = "Vancouver" AND ID < 100

OR

salary > 50000 AND city = "Toronto" AND ID < 100 ;

Q1: *salary > 50000 AND city = "Montreal" AND ID < 100 ;*

Q2: *salary > 50000 AND city = "Vancouver" AND ID < 100 ;*

Q3: *salary > 50000 AND city = "Toronto" AND ID < 100 ;*

To get the candidate set C of queries from the cache for any incoming query Q, the table involved in Q is identified first. All queries over the same table are selected from the cache. Having found all cached queries that match the table, attributes involved in the WHERE clause of Q are compared to the attributes in the WHERE clause of the cached queries so far selected. Only cached queries that have the same attributes in the WHERE clause are further considered. Then, for every cached query considered, the columns in the SELECT clause are compared to those in the SELECT clause of Q. The columns in the SELECT clause of Q need to be a subset of the columns in the SELECT clause of a cached query for the cached query to remain in the candidate set. If not, the cached query is not considered as a candidate query for Q. Thus, after comparing tables, attributes in the WHERE clause and columns in the SELECT clause, all the cached queries (C_i) so far selected are considered as candidate queries for potentially containing the response of Q.

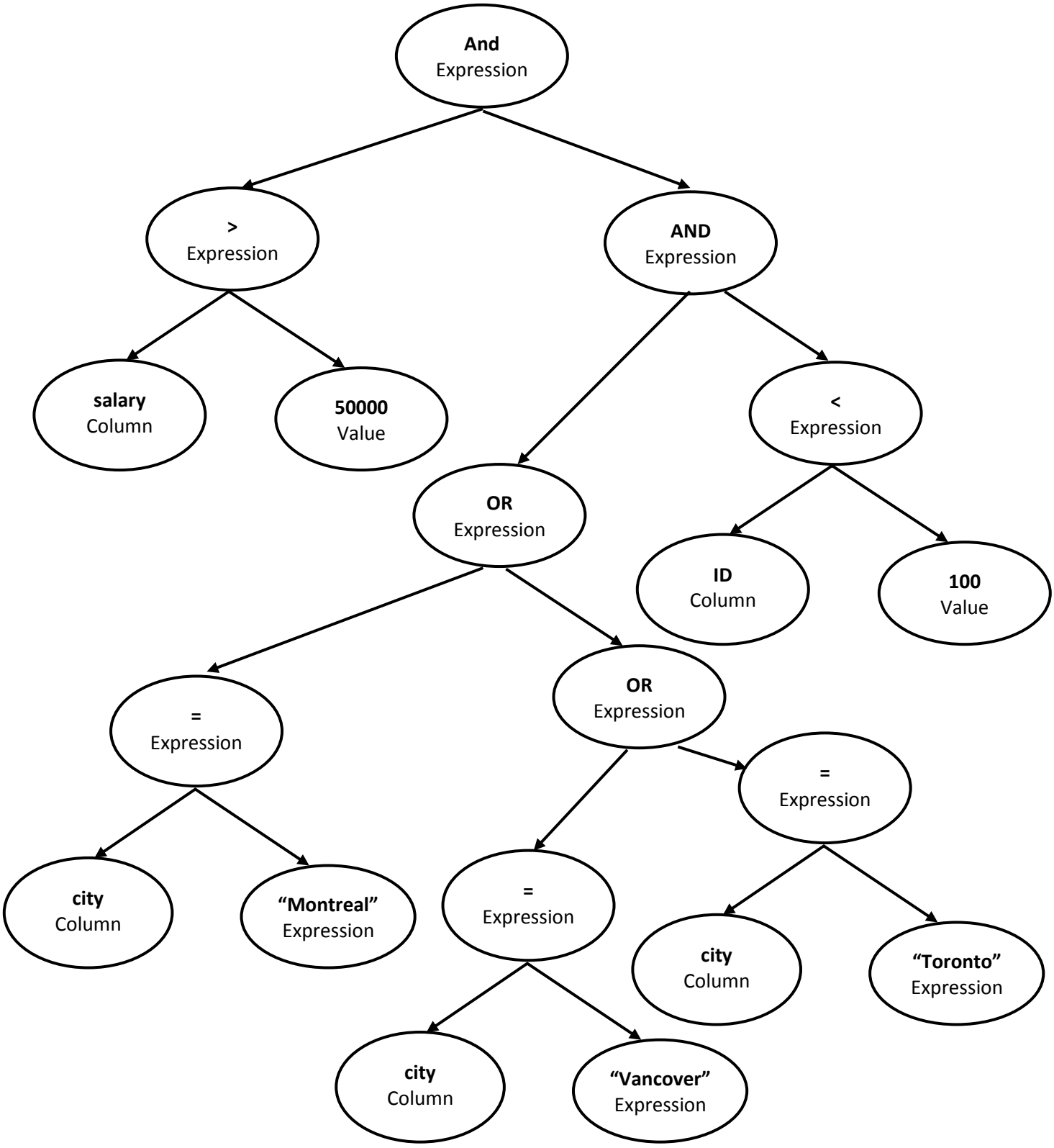


Figure 10: Tree shown for the WHERE clause of a query
*select * from employee where salary > 50000 and (city = "Montreal" or city = "Vancouver" or city = "Toronto") and ID < 100*

3.4 Range Checker

Basis for finding a full or partial cache response in the query cache is to identify the range for every inequality predicate. This is simply achieved by finding the upper and lower bound of the range. That is, for each inequality predicate in Q and all $C_i \in C$ we calculate the bounds. Then, for each C_i in C and for each inequality predicate in C_i we compare the bounds with the bounds of the corresponding predicate in Q. If the bounds of C_i are within the bounds of Q, C_i remains in the candidate set as a potential candidate query. Some example predicates and their corresponding ranges are shown below:

salary > 50000 and salary <= 80000 : **salary (50000 , 80000]**

salary >= 30000 and salary < 50000 : **salary [30000 , 50000)**

salary > 50000 : **salary (50000 , α**

3.5 Remainder Query Generation

An integral part of the Contquer algorithm is the generation of the remainder query. Once the candidate query $C_i \in C$ with the largest result set is determined in Step 5 of the algorithm, a remainder query needs to be generated to get the missing tuples. The Remainder query $R = Q - C_i$ is generated by rewriting the inequality predicates so as to get the tuples that are not covered by the partial cached query.

To better understand how a remainder query is generated, we take an example query (Query D) and a cached query (C) that contains partial results of Q.

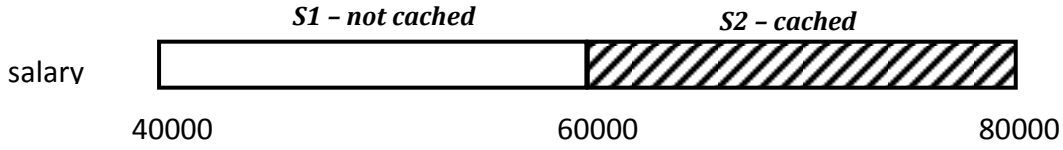
Q: *city = 'Montreal' AND salary > 40000 AND salary < 80000 AND age > 25;*

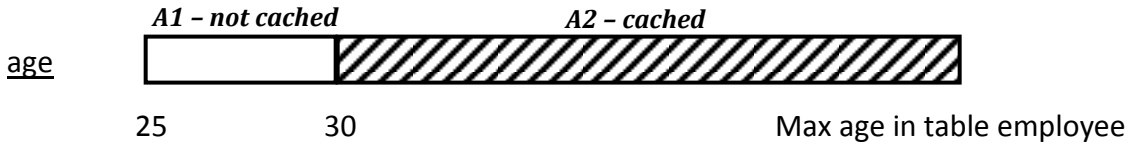
C: *city = 'Montreal' AND salary > 60000 AND salary < 80000 AND age > 30;*

In the above queries, there is one equality predicate (city attribute) and two inequality predicates (salary and age attribute). The ranges for the two inequality predicates are found through the range-checking algorithm:

Ranges of Q: salary (40000 , 80000) ; age (25 , α

Ranges of C: salary (60000 , 80000) ; age (30 , α





The upper bound for the age predicate for both Q and C is the maximum value of age in the table employee. From the ranges identified for Q and C, the ranges for the missing tuples (S1 for salary and A1 for age) are identified, and represented by a remainder query.

Ranges of R: salary (40000 , 60000] ; age (25 , 30]

Now there are two options to build remainder queries to fetch the missing tuples in C

in order to complete Q's response. In both cases, we have to make sure that the remainder query preserves all the equality predicates as is. In the example above, *city = 'Montreal'* is included in all the remainder queries generated. The approaches differ in how they determine the inequality predicates.

3.5.1 Approach # 1

In the first approach the first remainder query R1 is generated as follows by taking the missing S1 and A1 ranges:

R1: *city = 'Montreal' AND*
salary > 40000 AND salary <= 60000 AND
age > 25 AND age <=30;

But the remainder query R1's response combined with C's response still is a partial response to Q. As there are still missing tuples for salary between 40000 to 60000 (inclusive) and age greater than 30 and also for salary between 60000 to 80000 and age between 25 to 30 (inclusive). Therefore, R2 and R3 are generated.

R2: *city = 'Montreal' AND salary > 40000 AND salary <= 60000 AND age > 30;*

R3: *city = 'Montreal' AND salary > 60000 AND salary < 80000*
AND age > 25 AND age <=30;

We can illustrate the ranges of Q covered by C, R1, R2 and R3 in Figure 11.

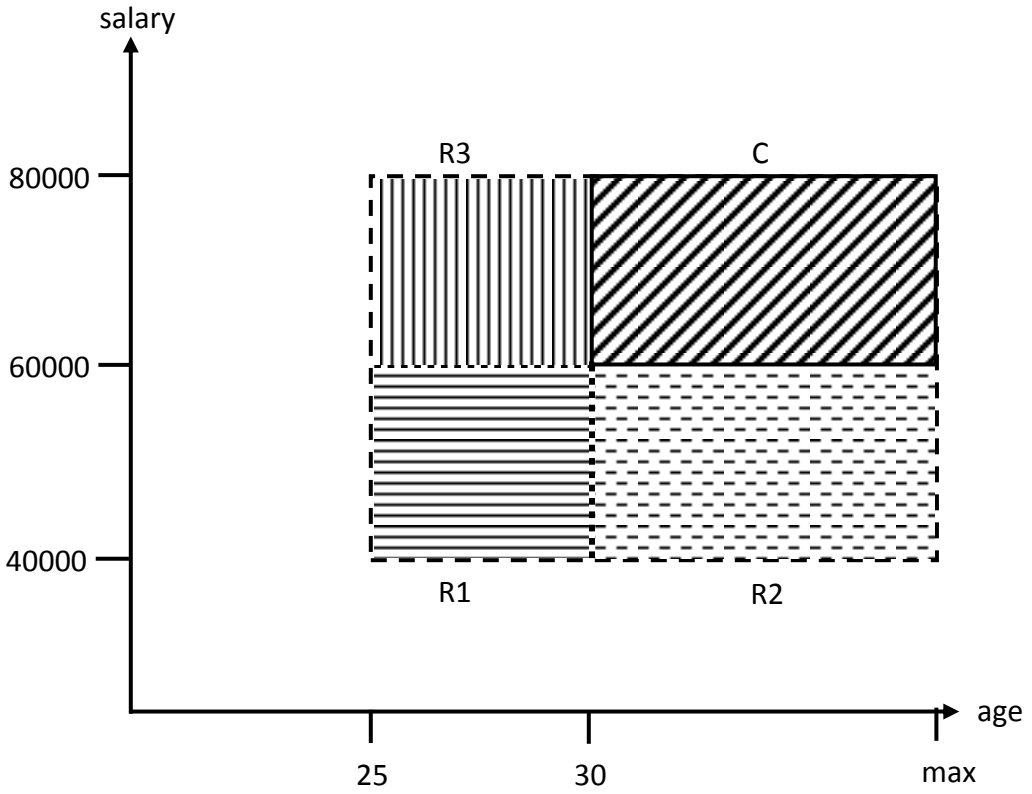


Figure 11: Graph plotting cached query C and remainder queries (R1, R2 and R3) for Approach1

In general, for a query Q with two inequality predicates, and a corresponding candidate query C, the generation of remainder queries can be described as follows:

Q: a_1 AND a_2

C: b_1 AND b_2

The remainder queries generated are:

R1: $(a_1 – b_1)$ AND $(a_2 – b_2)$

R2: $(a_1 – b_1)$ AND b_2

R3: b_1 AND $(a_2 – b_2)$

In Figure 12, each row of the two matrices represents a remainder query. The two matrices represent remainder queries generated for queries with two and three inequality predicates, respectively. The value 'C' for inequality predicate p1 in the matrix means that the corresponding remainder query has to have a value the same ranges as C for the predicate P1. The value 'd' indicates that the remainder query must take the range difference between Q and C for this predicate.

In general, for 'n' inequality predicates in a given query 2^n-1 remainder queries have to be generated.

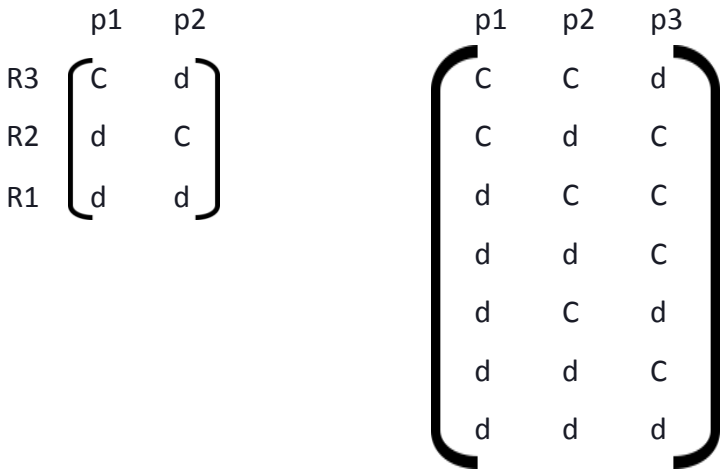


Figure12. Matrices representing remainder queries for two and three inequality predicates in a given query for Approach 1

3.5.1 Approach # 2

In the second approach, a remainder query is generated individually for every inequality predicate. Only one inequality predicate's range is changed to get the missing tuples, all other inequality predicates maintain Q's original range.

In the example above, we first consider the salary attribute and transform Q to get the missing tuples for salary ranges between 40000 to 60000. The age predicate is not changed and has the same value as in Q.

R1: *city = 'Montreal' AND*
salary > 40000 AND salary <= 60000 AND
age > 25;

Similarly, the other remainder query is generated by considering the age inequality predicate, transforming Q to get the missing tuples for age between 25 and 30 while the salary predicate has the same value as in Q.

R2: *city = 'Montreal' AND*
salary > 40000 AND salary < 80000
AND age > 25 AND age <=30;

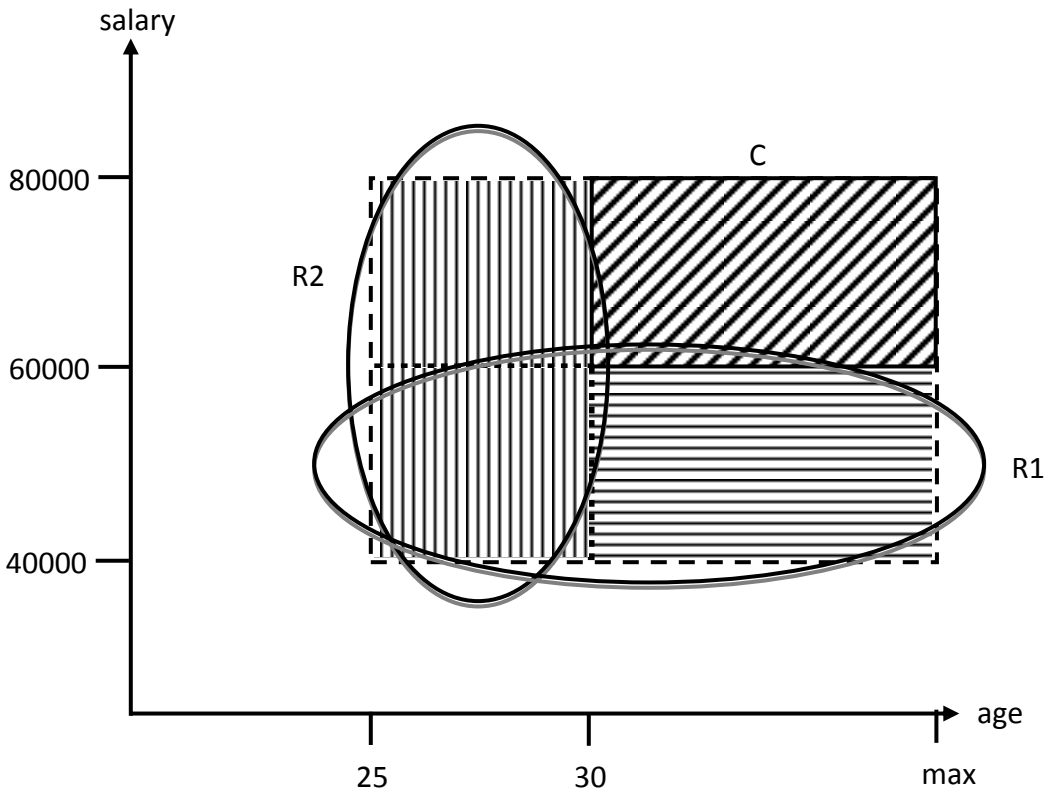


Figure 13: Graph plotting cached query C and remainder queries (R1 and R2) for Approach2

This approach, however, results in duplicate entries as seen in Figure 13. R1 and R2 have a region where their result sets overlap. When the result sets are merged in step 6 of the Contquer algorithm, these duplicates have to be eliminated, leading to computational overhead. On the other hand, considerably fewer remainder queries are generated.

For a query Q having two inequality predicates and a corresponding candidate query C, we can generalize as follows:

Q: a_1 AND a_2

C: b_1 AND b_2

The remainder queries generated are:

R1: ($a_1 - b_1$) AND a_2

R2: a_1 AND ($a_2 - b_2$)

In Figure 14, the value 'Q' for inequality predicate p1 in the matrix means that the remainder query generated has to have the same range as incoming query Q for predicate p1. Similarly the value 'd' indicates that the range difference (Q-C) must be taken for this predicate.

In general, for 'n' inequality predicates in a given query there are only n remainder queries generated through this approach.

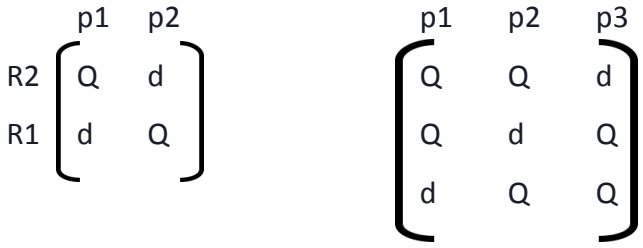


Figure 14: Matrices representing remainder queries for two and three inequality predicates in a given query for Approach 2

3.6 Final Example

Let us consider a further example to understand the working of Contquer algorithm. Consider the following incoming query (Q) and a candidate set C having five initial candidate queries C = {C1, C2, C3, C4, C5}. This initial set is obtained as described in Section 3.3. Only the WHERE clauses of the queries are shown in the example.

Q: city = “MONTREAL” AND salary >= 25000 AND salary < 80000

C1: city = “MONTREAL” AND salary > 60000 AND salary < 80000
C2: city = “TORONTO” AND salary >= 25000 AND salary < 60000
C3: city = “MONTREAL” AND salary >= 25000 AND salary < 45000
C4: city = “VANCOUVER” AND salary > 60000 AND salary < 100000
C5: city = “MONTREAL” AND salary > 100000 AND salary < 150000

The Contquer algorithm first of all checks if the queries in C fulfill all the equality predicates of Q. C2 and C4 do not fulfill this condition and are removed from the list C and eliminated from further consideration. The updated candidate set is C = {C1, C3, C5}. The next step determines for all inequality predicates the ranges of all candidate queries in C and of Q. Only the salary attribute has to be considered in our example.

Q: salary [25000 , 80000)
C1: salary (60000, 80000)
C3: salary [25000, 45000)
C5: salary (100000, 150000)

After identifying the ranges, the algorithm checks for containment by comparing the ranges of each of the queries in the candidate set C with the corresponding ranges in

Q. C1 and C3 are within the range of Q, and therefore, have potential for being selected for partial containment. C5 is not in the range of Q and is discarded. The updated candidate set is now C = {C1, C3}.

As discussed previously, if there is more than one potential query to be selected, the query in C having the most tuples in its result set is selected. Assume C3 has more tuples in its result set than C1. Then C3 is selected as a partial hit and its result set is saved for a merge later on. After C3 is selected, C = {C1} and a remainder query (R = Q – C3) is generated:

R: city = “MONTREAL” AND salary >= 45000 AND salary < 80000

A recursive algorithm is employed. The remainder query R becomes the current incoming query Q and the Contquer algorithm is continued with the current candidate set which only has C1 left in the candidate cached query list. Therefore, the range for inequality predicate for both C1 and Q are compared.

Q: salary [45000 , 80000)
C1: salary (60000, 80000)

C1 is within the range of Q and a partial response can be gathered from it and is considered as a partial hit. A new remainder query is generated as (R = Q – C1):

R: city = “MONTREAL” AND salary >= 45000 AND salary <= 60000

Since there are no more queries in the candidate set that can be compared for overlapping regions, the final remainder query is sent to the database system. Finally, the results from partial cached queries of C3 and C1, and the tuples from database are

merged and returned to the client.

3.7 Complex Queries

Queries can have special operators in their WHERE clauses such as IN and BETWEEN. The IN operator is handled similar to OR operators. For that, IN operators are transformed into OR operators. For example the following query uses the IN operator in its WHERE clause, which we transform into an OR operator.

```
SELECT *                                    SELECT *
FROM employee                               FROM employee
WHERE city IN (‘Montreal’ , 'Toronto')  ⟶    WHERE ( city = 'Montreal'
      AND salary > 40000;                           OR
                                                    city = 'Toronto' ) AND
                                                    salary > 40000;
```

BETWEEN operators are transformed into two inequality comparison predicates of greater than equals (>=) and lesser than equals (<=) for the same attribute. For example the following query, using the BETWEEN operator in its WHERE clause, is transformed so that it only has inequality predicates.

```
SELECT *                                    SELECT *
FROM employee                               FROM employee
WHERE salary BETWEEN                  ⟶    WHERE salary >= 40000 AND
      40000 AND 80000;                              salary <= 80000;
```

3.8 Commutativity of Predicates

AND and OR operators are commutative. The order of the predicates does not change the semantics of the query and its result set. For example, the following three queries are syntactically different but all of them have the same response.

Q1: salary > 40000 AND salary < 80000 AND city = ‘Montreal’;

Q2: city = ‘Montreal’ AND salary > 40000 AND salary < 80000;

Q3: city = ‘Montreal’ AND salary < 80000 AND salary > 40000;

Since in this thesis we only look for single table queries and all the queries are transformed into DNF, supporting commutativity is simple.

Chapter 4

Distributed Cooperative Query Caching Architecture

4.1 Introduction

In the previous chapter we introduced how the Contquer algorithm works on a single application server. This chapter is dedicated to a distributed environment. We first present a distributed caching infrastructure that allows for cooperation of caches. Then we show how Contquer can be used in this distributed environment.

4.2 Distributed Cooperative Cache

To meet the demands for large-scale web-sites, it is very common to use multiple application servers. A load balancer forwards each HTTP request to one of these application servers, for example, in a round robin fashion. As each application server has its own cache, we have multiple caches in the system. This can reduce the load on

the database if utilized effectively.

In most existing systems, each application server caches data without knowing what other application server caches have stored. With such an approach it is possible that the same object resides in many caches. As a result, the full capacity of the cache space is not exploited.

Therefore, in the context of this thesis work we have developed a cooperative caching infrastructure. Each application server cache in the system has knowledge of all caches and has access to contents of other caches. As a result we can utilize the aggregate capacity of all the caches very well. We term this approach "distributed cooperative cache". That is, every cache in the system knows what other caches have stored. Thus, if a request for a certain query or object is made to an application server that does not have that query or object in its cache it can check if other caches have that requested query or object. If this is the case, a remote call is made to the remote cache fetching the object from there.

In order to facilitate such remote access each cache needs to know the content of the other caches. We have implemented this distributed cooperative cache with the JBoss/Hibernate/EHCache infrastructure as shown in Figure 15.

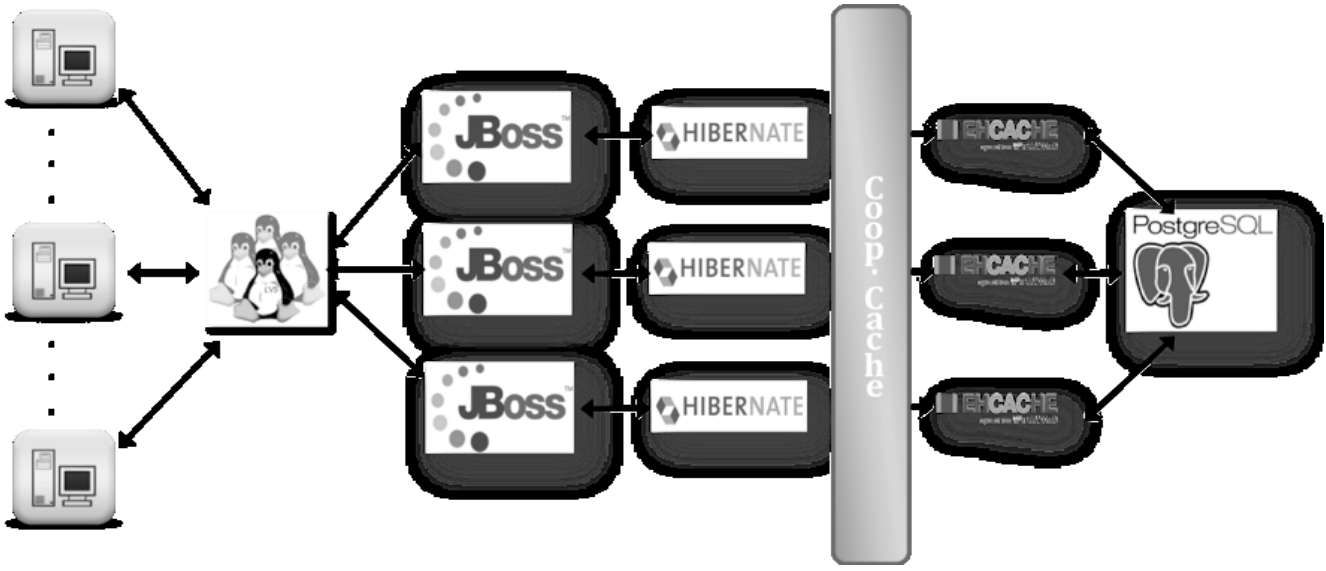


Figure15: Our cooperative cache layer in JBoss / Hibernate / EHCache infrastructure

Our cooperative cache is a thin layer above EHCache, the second-level cache. It wraps the original EHCache providing exactly the same interface. Whenever Hibernate makes a call to check whether a data item is in the second-level cache our cooperative layer will either return it from the local cache or a remote cache if it resides in any of the other caches.

In order to know where objects are cached, the cooperative cache layer at each node maintains a cache directory. The directory contains for each object cached a reference to the location of the cached object. The directory information also keeps track of the content in the query cache. For example, an object O stored at application server A would be listed as a local object available locally from its second level cache but for all other caches in the system, O would be listed as a remote object located at application server A. On receiving a request for O, if the request is received by application server A, a local call is made to its cache. Otherwise, if it exists remotely, the location in the

directory structure helps us identify which cache has the object and therefore, a remote call is made to that cache to fetch O.

Having such cooperative mechanism in a distributed environment, it is important that, if possible, we utilize the aggregate cache capacity of the overall system as much as possible. Thus, we make sure that any object is stored only once in the whole system. If an object O exists at application server X, none of the other application servers' caches in the system will put O in their local caches. This greatly increases the overall cache capacity. The cache directory, distinguishing between local objects and remote objects, makes sure that an object is cached only at one location.

A query might not exist in the query cache but it is possible that some of the objects in its result set exist in either local or remote second-level caches. As we discussed, we make sure only one copy of an object exists in all the caches. So irrespective at which node the query is requested, objects in the query's result set that do not exist in any of the second-level caches are fetched from the database and cached at that requested node's second-level cache.

The cache directory up-to-date at each server needs to be kept up-to-date. Whenever a new object/query is loaded into a cache or an object/query is removed from a cache, the cooperative cache layer at this node multicasts this information to all other servers who then update their directory information. As a result, the directory information at all servers is consistent.

As an example, assume that client X poses a query Q1 at application server A (e.g. all products priced less than $2000). Also, assume the query Q1 is not cached yet. Thus, it is submitted to the database and the results are cached in both the first level cache and the second level cache. This indicates the query string and associated object

identifiers in the query cache and objects themselves are stored in the second-level cache.

Furthermore, the information that query Q1 and its result set are newly cached at application server A's cache is published to all the application servers in the whole system. Therefore, the cooperative caching layer at all servers update their directory information to list Q1 as cached at application server A. This means the directory at application server A contains a reference for Q1 pointing to its cache entry in the query cache. The location is marked as local since it is locally stored in its own cache. The cache directories at all other application servers in the system, however, only have a reference of Q1, i.e., the query string of Q1 and application server A listed as its location. Thus, if a client Y poses the same query Q1 at application server B, the cooperative caching layer at B will find in the cache directory that query Q1 is already cached at application server A. Thus, the cooperative cache layer makes a remote call to A, fetches the identifiers of all objects in the result set of Q1. The objects themselves might be located at different application servers' second-level caches. In order to have fast object transfers, we make batch calls to each server that has at least one cached object. This batch call retrieves all objects in the result set that are cached by this server with only one data transfer. Upon merging results from various application servers, the final result set for Q1 is formulated and is returned to the upper layer.

4.3 Contquer in a Distributed Environment

The Contquer algorithm supports the distributed cooperative caching model as discussed in Section 4.2. This means that all the properties of the distributed cooperative caching model can be used in the query caching model that we have developed.

As objects are distributed across different caches in the distributed cooperative caching model, queries and their results are distributed across different caches. However every cache knows about all the queries that other cache nodes have stored, but not their result sets.

Thus, in the algorithm, when identifying an initial candidate set, all queries that are cached in the whole system are taken into consideration. The cache directory that exists in every application server contains all the query strings that are either cached locally or remotely. The cache directory therefore provides us with all cached queries, from which an initial candidate set is identified. The Contquer algorithm then looks for exact matches, and if not found, does further confinement of candidate sets as explained before. If a candidate selection is required for partial containment from a candidate set having more than one query, the one with the largest number of tuples should be selected. In this case if the query in the candidate set is located at a remote location, a remote call is made to get the number of tuples for that query.

Chapter 5

Experimental Results

5.1 Introduction

This chapter shows results from the experiments conducted to analyze the behavior of the cooperative cache model and the Contquer algorithm. We begin with a description of the performance metrics used in our performance measurements, and the benchmarks that were used for the experiments. This is followed by a comparison of performance numbers for different caching designs.

5.2 Hardware Platform

All machines, which include client emulator, application servers and the database, have the same hardware. Each machine has Intel (R) Pentium (R) Dual CPU 2.80GHz, 1 GB RAM and an 80 GB hard disk drive. All machines are connected through a 100 Mbps LAN switch.

5.3 Software Environment

All machines run Ubuntu 9.1 Linux distribution and have three application servers running on three different machines. We use JBOSS 5.1.0 as our application server. Linux Virtual Server (LVS) installed on a different maching has been used as a load balancer. All the HTTP requests go through LVS to the 3 application servers in a round robin scheme. PostgreSQL 8.4.1 is used as our back end database server, again installed on a different machine.

5.4 Performance Metrics

We use several performance metrics to analyze the behavior of the caching system.

5.4.1 Average Response Time

We measure average response time as:

T = total response time for n queries / n.

It represents the total time needed from sending a web-page request (HTTP request) from the client to receiving the response and showing it on the client's browser.

Note that we only consider clients in the same LAN.

5.4.2 Throughput

Throughput is measured as web-interactions per second (WIPS), i.e., the total number of web requests that are satisfied per second by the application servers.

5.4.3 Query Cache Hit Rate

The query cache hit rate includes both full and partial hits, and is represented as two different entities. Any query that has an exact match or is fully contained in the query cache is considered a query hit. Otherwise, any incoming query having one or more partially contained queries in the cache is considered as one partial hit, irrespective of the number of partial responses contained in the cache for that incoming query.

5.5 Measurement Methodology

We evaluated our Contquer model using two different benchmarks, namely RUBiS and a micro-benchmark. We use a client emulator, which allows us to change the load of the application by varying the number of emulated clients. Each experiment runs on an average for one hour, where the first 25% of the time is used to warm up the cache (warm-up phase) and the last 15% of the time is considered as cool-down phase. The time in between is considered as measurement phase and the results are measured only in this phase. Each experiment also starts with the same identical database.

We compare EHCache, the cooperative cache model and the Contquer implementation. EHCache represents the normal caching mechanism including simple query caching. Cooperative cache is the distributed caching mechanism where duplicate entries in the cache are avoided. It also includes a simple query caching model. The Contquer model represents the implementation of our Contquer

algorithm, which is again a distributed caching model.

5.5.1 Load Generation

Workload is generated in the used benchmarks by having multiple clients running concurrently. Each client sends request to the application servers as soon as the previous request's response is received.

5.5.2 RUBiS Benchmark

RUBiS is an auction site prototype modeled after eBay that is used to evaluate application server performance and scalability [16]. Several versions of RUBiS are available. We have used the RUBiS Hibernate Java servlets version for our experiments. The benchmark is purely read-oriented, as we have only used the browsing mix of the benchmark for the evaluations.

5.5.3 Micro Benchmark for Query Caching

In RUBiS, most of the queries can only be satisfied by an exact query as queries mostly have only equality comparison predicates and no inequality comparison predicates in there WHERE clause. Thus, we needed another benchmark that can reflect the true potential of the Contquer cache model and act as our primary benchmark.

The micro-benchmark uses the same database schema as the one used in RUBiS. Based on that schema a few queries were generated that included queries having equality predicates only, and also both equality and inequality predicates. The generated queries had empty query parameters and for each of these queries many different queries were generated having different values of the parameters. The

parameter values were assigned randomly but lied within the minimum and maximum values that existed in the database for that attribute. For example a query Q was generated as follows:

Q: *SELECT **
FROM item
WHERE category = ? AND price > ? AND price < ? ;

The query parameters, i.e., the attribute values of category and price were left empty. Starting from there, multiple queries were generated with assigned attribute values such as (only WHERE clause shown here):

Q1: category = 4 AND price > 1000 AND price < 5000
Q2: category = 5 AND price > 3000 AND price < 4500
Q3: category = 6 AND price > 100 AND price < 1000
Q4: category = 4 AND price > 3000 AND price < 4000
Q5: category = 4 AND price > 7500 AND price < 10000

Above queries (Q1 to Q5) are only a few queries that were generated from the source query Q.

5.6 Experimental Results

We studied the effects of the Contquer caching model with the two benchmarks separately.

5.6.1 RUBiS Benchmark

As discussed, most of the queries in this benchmark are very specific in nature and most include only equality predicates in their WHERE clauses. Still, there were a few queries that could potentially exploit the Contquer algorithm.

Figures 16 and 17 show the average response time and throughput for the browsing mix of RUBiS, respectively, with 1000, 1500 and 2000 clients submitting the requests.

Compared to EHCache, the cooperative cache has significant lower response time and high throughput. Although it only supports simple query caching, the fact that the cooperative cache can hold more objects as caches can cooperate with each other, more queries can be stored in the cache (local or remote). Therefore queries can be served more often from the cache leading to lower response time and high throughput.

The Contquer cache model reduces response time and increases throughput further, but to a lesser degree. The reason for the little improvement is that there are not many queries in the benchmark that can potentially become partial hits.

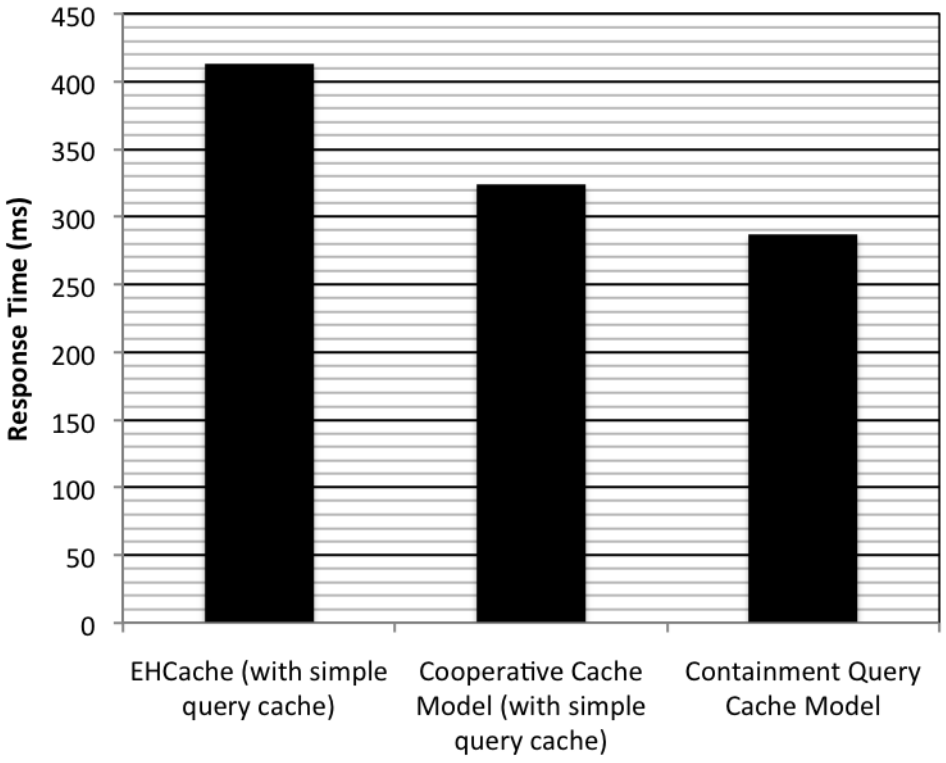


Figure 16: Average response time for RUBiS benchmark

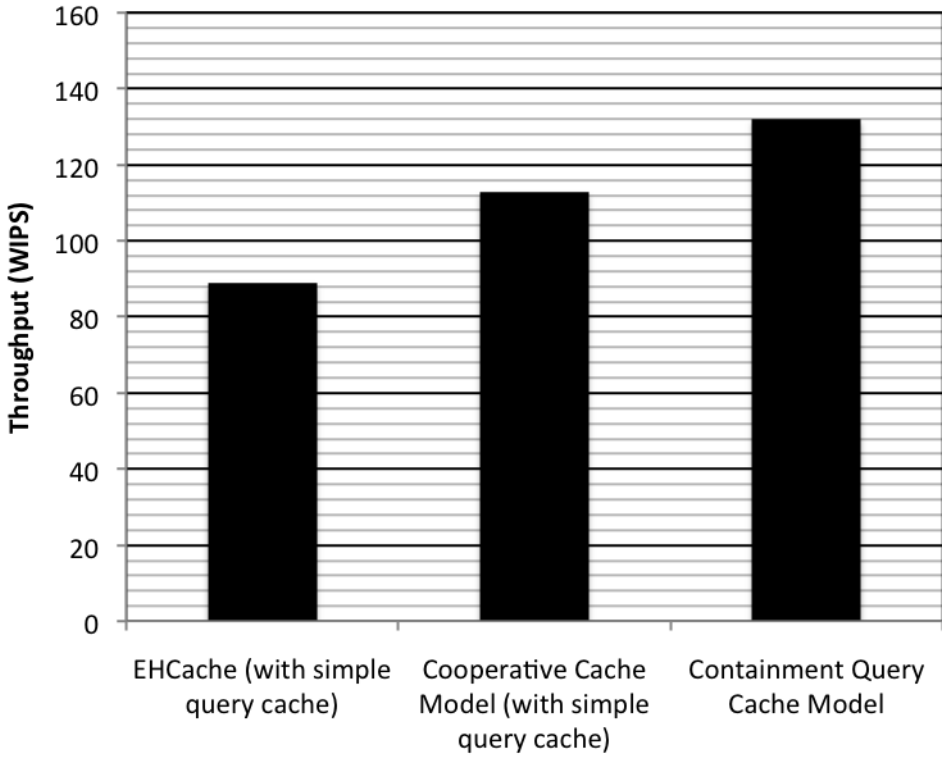


Figure 17: Throughput in WIPS for RUBiS benchmark

The results in response time and throughput are best explained by looking at cache hit rates. The gain in query cache hit rate as shown in Figure 18, is very substantial for the cooperative cache compared to EHCache as it improves from 23% to 48%, almost 110% improvement. Incoming queries forwarded to an application server may be cached at some other application server and therefore, because of the cooperation between the nodes, it results in a query hit, which is not the case for EHCache. For the Contquer cache model the query hits increased from 48% to 50%, which is a very small change. Here, the 2% change only reflects queries that are fully contained within the query cache. A 7% further improvement results from partial containment of cached queries, which results in an overall improvement of 20% for the query hits. Note, that the increase in the query hits due to partial containment means that 7% of all queries have one or more partially contained queries, which also resulted in generating remainder queries that are sent to the database.

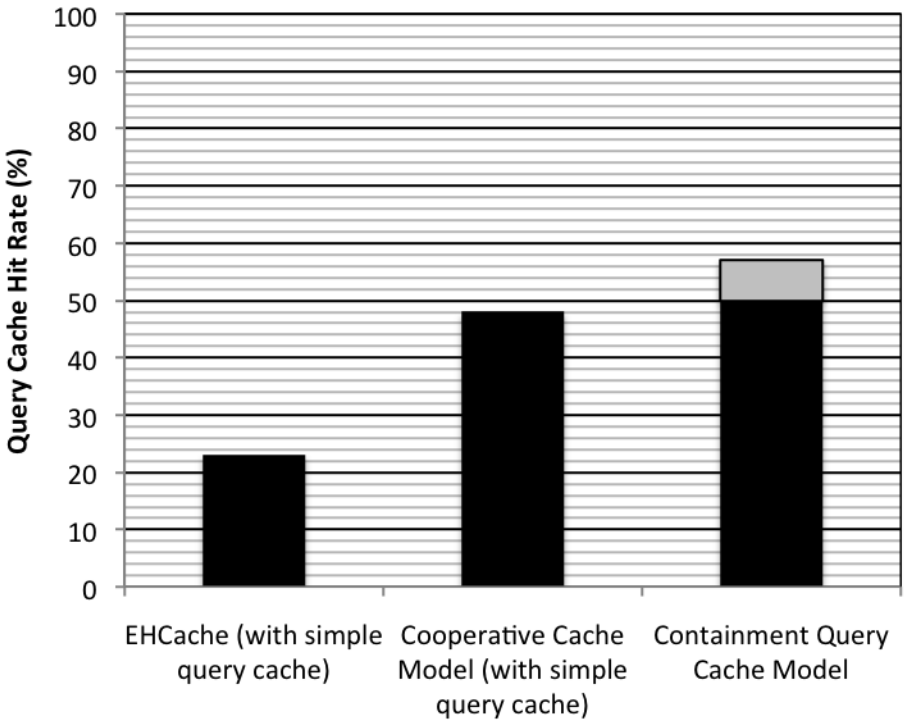


Figure 18: Query cache hit rate (%) for RUBiS benchmark and partial hits for Contquer cache model

5.6.2 Micro-Benchmark

The micro-benchmark acts as our primary benchmark as it reflects the true potential of the Contquer algorithm. First, we wanted to figure out the breakdown of response time in terms of percentage of time spent at the database and the application servers. Figure 19 shows this breakdown of response time for EHCache, the cooperative cache and the Contquer cache model. We clearly see that for EHCache, most of the time is spent at the database, showing the huge potential for performance gains from query caching. In fact, this gain in performance improvement can be seen by looking at the results for the cooperative cache and Contquer cache model.

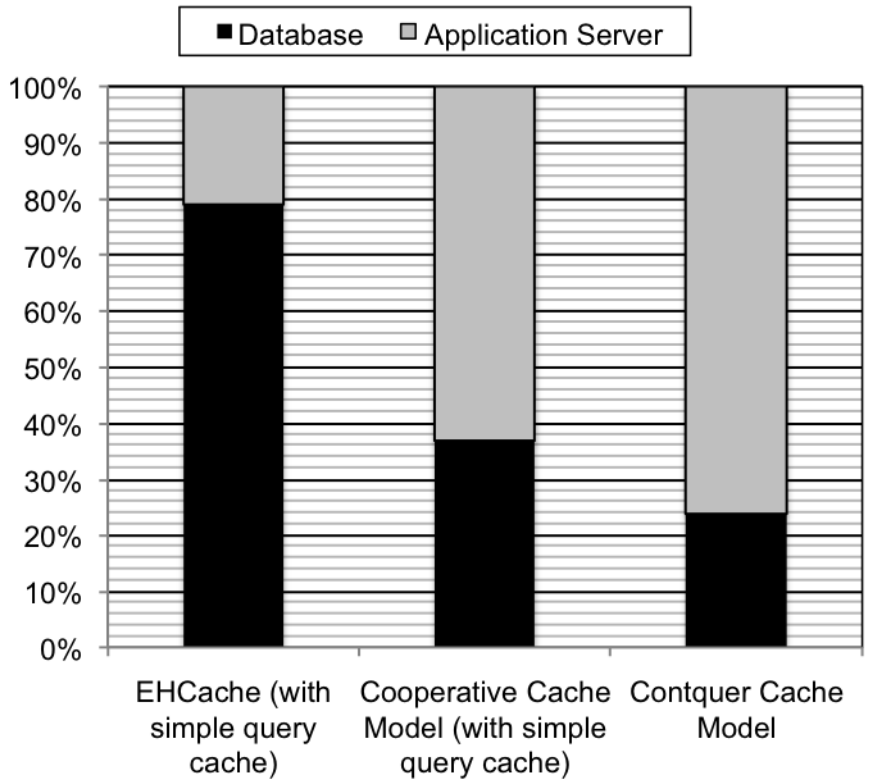


Figure 19: Breakdown of the response time into percentage of the total time spent at the database and application server

In Figure 19, the total time spent at the database represents the total time of sending the query to the database from the application server, execution of the query at the database and the results returned to the application server. The percentage of the total time spent at the application server includes:

- Network traffic generated between the client and the application server including the HTTP request sent from a client to an application server and the response that is sent back to the client by the application server in the form of HTML pages.

- Application server processing.

- Network traffic among application servers due to cooperation. This occurs only in the cooperative cache and Contquer cache model.

Figures 20 and 21 show the response time and throughput, respectively, for the micro benchmark for the different caching models. Since the database becomes the bottleneck during some periods of the experiment, the cache improves throughput, in addition to producing better response times. The response time with the cooperative cache is barely half the response time recorded for EHCache. The Contquer model further reduces response times in a significant way. Similarly, throughput is significantly improved from 65 WIPS for EHCache to 154 WIPS (almost 140%) for the cooperative cache model and to 181 WIPS for the Contquer cache model.

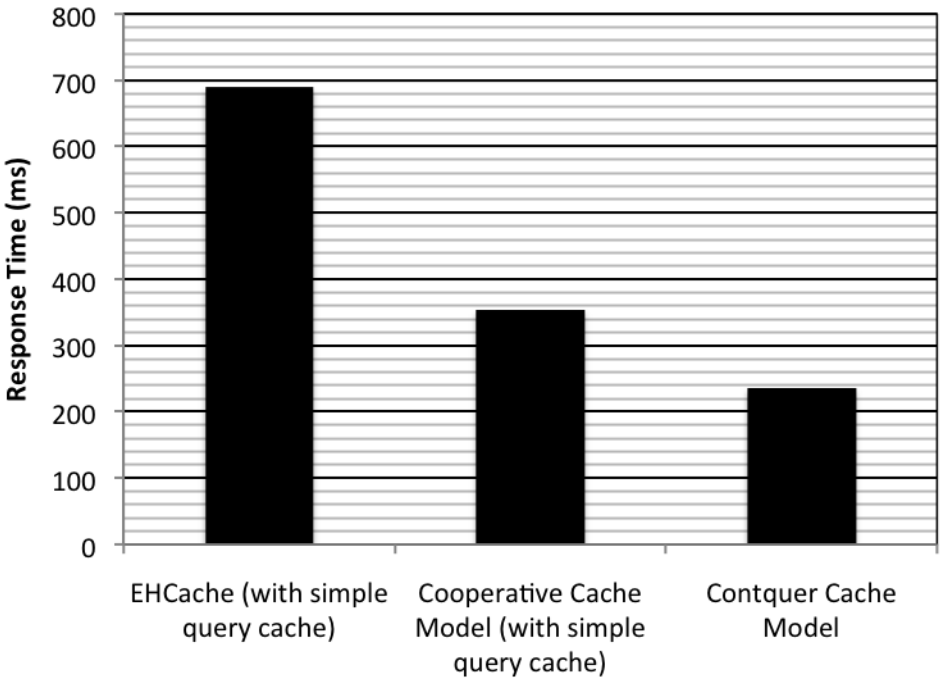


Figure 20: Response time for micro benchmark

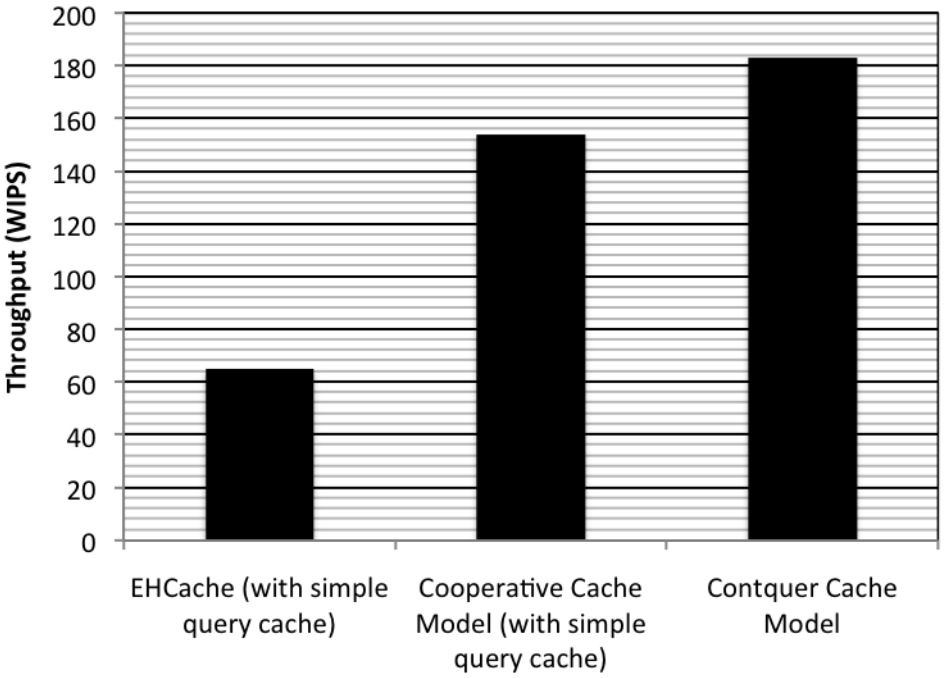


Figure 21: Throughput (WIPS) for micro benchmark

The query hit rates for EHCache, cooperative cache and Contquer cache model are shown in Figure 22. The improvements are significant for both the cooperative and Contquer cache model. Almost 240% improvement compared to EHCache is recorded for the cooperative cache, which had 44% query hits as compared to 13% for EHCache. For the Contquer cache model, the full query hits increased to 51% and further improvements resulted from 15% partial hits of cached queries. The increase of 7% (44% to 51%) in cache hit rates means that the Contquer cache model found a full containment from more than one query. The 15% partial hits are further categorized in Figure 23. 4.2% of the overall queries found one matching sub query in the cache, 4.8% found two, 3.15% found three, 1.2% found four and 1.65% found five or more partial queries in the cache.

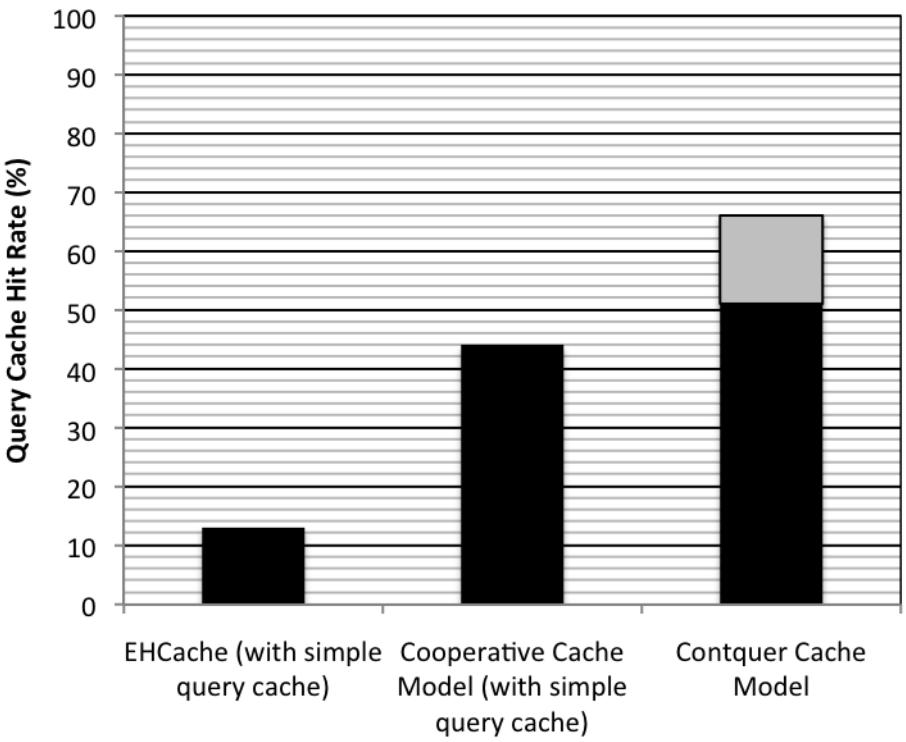


Figure 22: Query cache hit rate for micro benchmark

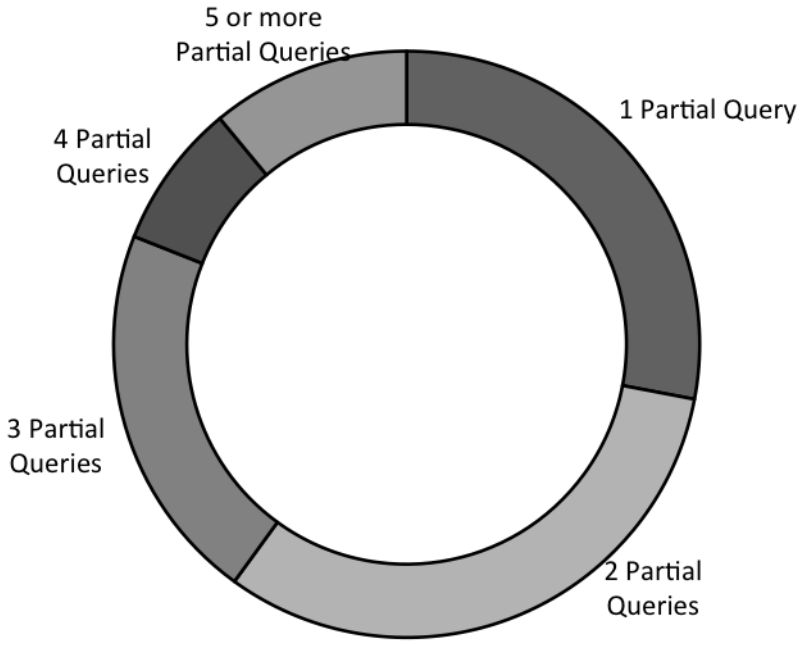


Figure 23: Partial hit rate breakdown into number of partial queries found in the cache

Finally, Figure 24 shows the response time for different number of clients. The experiment was conducted under different number of emulated clients for EHCache, Cooperative cache and Contquer cache model. Response time becomes stagnant upon reaching 7500 clients for all the models. Upon heavy load of over 9000 clients the experiment saturates, thereby increasing the response time drastically. Although the change in response time from 8500 to 9000 clients for EHCache model is more than our cooperative cache model and Contquer cache model.

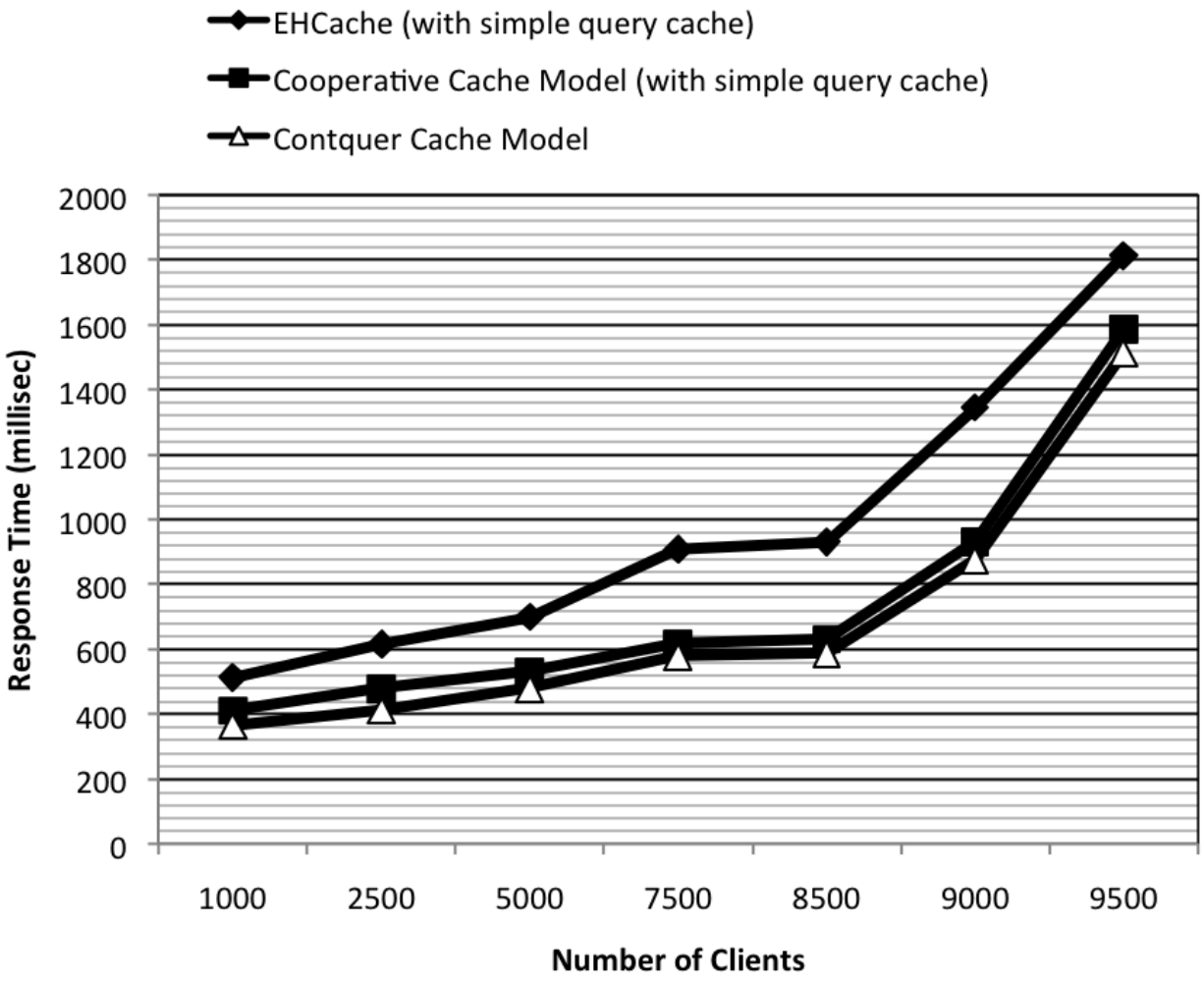


Figure 24: Performance comparison for different number of clients for micro-benchmark

Chapter 6

Conclusions and Future Work

6.1 Introduction

In this chapter we discuss the contributions made by this thesis. We start with an overview of the driving principle in the design of our cooperative query caching algorithm Contquer. We then discuss the contributions made by this thesis. Then, we briefly discuss the different steps of the Contquer algorithm. We conclude with a discussion of results on two different benchmarks that were used to conduct all the experiments for measuring performance improvements. In the future work section, we discuss possible enhancements that can be done to Contquer.

6.2 Conclusions

E-commerce web applications are hosted in a multi-tier architecture, where database systems are often the bottleneck. A common approach to scale the database component is query result caching. Queries and their results are cached such that future requests for the same queries do not need to go to the database, and have shorter response times. Work already exists in this area but has two major limitations.

First, the aggregate capacity of all the caches in the system is not fully exploited. Second, queries can only be served from the cache if they exactly match an exisiting query.

In this thesis we introduce Contquer, a distributed cooperative query caching algorithm. It uses a distributed cooperative caching architecture whereby all application server caches interact with each other such that an object is stored only once in the whole system. Therefore, we exploit the aggregate capacity of all caches. Local and remote calls are made to fetch objects that exist on local or remote caches, respectively. Moreover, we optimize the algorithm by serving not just exact queries but also part of the query's response from various partial queries. Missing records are fetched from the database by generating remainder queries.

The Contquer algorithm involves six steps. In the first step, the incoming query is transformed into a DNF. An initial candidate set for full or partial containment is identified as the next step. The third step looks for an exact match of the query in the cache. If an exact query is not found, refinement of the candidate set takes place as the next step. Equality and inequality constraints are checked. In step 5, a query is chosen which contains a partial response to the query posed. A remainder query is generated to get the missing tuples that do not exist in that partial response. The algorithm further looks for containment with this new remainder query until there are no partial queries in the cache. In that case, the missing tuples are fetched from the database. Finally, in step six all the results from different partial queries and database are merged.

We evaluate our cooperative caching model and Contquer with two different benchmarks. All the results show that our cooperative caching architecture improves response time and throughput because of a higher cache hit rate. Also, utilizing a

partial response for a query from the cache means lower load on the database.

6.3 Future Work

In this thesis, we only looked into single table queries, as queries involving multiple tables has significant complexity. The Contquer algorithm also does not support all SQL operators. Thus, queries with multiple tables are interesting and supporting operators like NOT, NOT IN, LIKE tasks for future work.

Bibliography

[1] C. Amza, G. Soundararajan, and E. Cecchet. Transparent caching with strong consistency in dynamic content web sites. In International Conference on Supercomputing (ICS), pages 264–273, 2005.

[2] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. In ACM Transactions on Database Systems (TODS), 22(3):315–363, 1997.

[3] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In ACM SIGCOMM Conference, pages 254–265, 1998.

[4] S. Gadde, M. Rabinovich, and J. S. Chase. Reduce, reuse, recycle: An approach to building large internet caches. In Workshop on Hot Topics in Operating Systems, pages 93–98, 1997.

[5] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In Proceedings of IEEE INFOCOM'99, pages 294–303, March 1999.

[6] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Krithi Ramamritham, and Dan Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In Proceedings of the 27th International Conference on Very Large Databases, pages 667–670, September 2001.

[7] Q. Luo, S. Krishnamurty, C. Mohan, H. Pirahesh, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In Proceedings of the 2002 ACM International Conference on Management of Data, pages 600–611, June 2002.

[8] Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In Proceedings of the 27th International Conference on Very Large Databases, pages 667–670, September 2001.

[9] http://dev.mysql.com/doc/refman/5.1/en/query-cache.html

[10] http://ehcache.org/

[11] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In ACM SIGMOD Conf., pages 177–190, 2002.

[12] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In ACM SIGMOD Conf., pages 532–543,

2001.

[13] M. E. Dick, E. Pacitti, and B. Kemme. Flower-cdn: A hybrid p2p overlay for efficient query processing in CDN. In Int. Conf. on Extending Database Technologies (EDBT), 2009.

[14] O. D. Sahin, S. Antony, D. Agrawal, and A. E. Abbadi. Probe: Multi-dimensional range queries in p2p networks. In Int. Conf. on Web Information Systems Engineering (WISE), pages 332–346, 2005.

[15] K. Lillis and E. Pitoura. Cooperative xpath caching. In ACM SIGMOD Conf., 2008.

[16] http://rubis.ow2.org/

[17] http://en.wikipedia.org/wiki/Disjunctive_normal_form