Semantic caching framework towards FPGA acceleration for range query processing in domain of massive distributed data

Huu van Long Nguyen

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Van Long NGUYEN HUU

Contribution à l'accélération FPGA de cache sémantique pour le traitement des requêtes d'intervalles dans le domaine des masses de données

Semantic caching framework towards FPGA acceleration for range query processing in domain of massive distributed data

Thèse présentée et soutenue à Lannion, le 02 Décembre 2022

Unité de recherche : IRISA, B<>Com and Nokia Bell Labs

Thèse N° :

Rapporteurs avant soutenance :

Claudia RONCANCIO	Professeur des Universités, Université Grenoble Alpes
Nicolas GAC	Maître de conférences, HDR, Université Paris Saclay

Composition du Jury :

	Claudia RONCANCIO	Professeur des Universités, Université Grenoble Alpes
	Nicolas GAC	Maître de conférences, HDR, Université Paris Saclay
	Karine ZEITOUNI	Professeur des Universités, Université de Versailles Saint-Quentin-en-Yvelines
Co-dir. de thèse :	Laurent d'ORAZIO	Professeur des Universités, Université de Rennes 1
Co-dir. de thèse :	Emmanuel CASSEAU	Professeur des Universités, Université de Rennes 1
Co-encadrant :	Julien LALLET	Docteur, Ingénieur de recherche, Nokia Bell Labs

Author

Nguyen-Huu Van Long was born in Can Tho, Vietnam. He obtained his engineer degree of computer science from Can Tho University in 2010. Since then, he has worked as a lecturer at Can Tho University. In 2014, he received his master degrees in embedded system from Université de Science et Technologies de Hanoi and Université de Limoges. He commenced his PhD in computer science at Université de Rennes 1 from January, 2020. His main research interests include database, distributed systems, caching mechanisms, and acceleration with commodity hardware.

To my family

Acknowledgement

First of all, I would like to express my deepest gratitude and appreciation to my advisors, Prof. Laurent d'Orazio, Prof. Emmanuel Casseau and Dr. Julien Lallet, for providing continuous support, guidance, mentoring throughout my PhD. They first gave me a great opportunity to pursue my research career in France. They have trained me how to do research properly and they were so patient with me as I was progressing slowly. My advisors always kept me motivated with their excitement for all works that I made. Needless to say that I could not do well my thesis without my advisors. Thank you very much for all!

I would thank Prof. Daniel Chillet and Prof. Farouk Toumani, for accepting to be the CSID Committee Members of my thesis. Their insightful suggestions and comments have motivated me to improve and complete my research.

My PhD would not be possible without the financial supports from the agreement between ANRT and Nokia Bell Labs, France in terms of CIFRE scholarship. I would like to acknowledge them for funding my research. Thank you all colleagues of Nokia Bell Labs and B<>Com for giving me a great working environment as well as kindly supports during Covid-19 pandemic. In addition, thank you Can Tho University, especially College of Information & Communication Technology, my workplace in Vietnam, that facilitates me to complete my research abroad.

Last but not least, I would like to thank my wonderful family for always being therefor me. I would like to thank my lovely wife, Thi-Thanh-Ha Nguyen, for her great understanding. Thank you my friends for supporting my daily life during three years in Lannion. All of them have always believed in me and supported me endlessly. Thank you! You are the main reason I am where I am today.

Declaration

This dissertation has been completed by Van Long Nguyen Huu under the supervision of Professor Laurent D'Orazio, Professor Emmanuel Casseau and Dr. Julien Lallet and has not been submitted for any other degree or professional qualification. I declare that the work presented in this dissertation is entirely my own except where indicated by full references.

Abstract

In recent years, we have witnessed a paradigm shift in analytical database systems over large and scalable resources. In particular, as an alternative to relational database management systems (RDBMS) with local data storage, there is an emergence of new data management systems (DMS) with remote and distributed data storage with respect to big data and cloud computing. More precisely, such systems can employ a layer or dis-aggregated storage model, where the elastic *compute layer* accesses data on an independently scalable *remote storage*. Given the relatively high latency in communication with respect to low bandwidth between these above layers, we consider that caching data at the compute layer now has become more important. Indeed, a cache system can increase fine-grained re-usability of data to mitigate unnecessary query re-executions. In summary, we are witnessing a renewed spike in caching technology for these DMSs where the hot data is kept at the compute layer in fast local storage of limited size to accelerate query processing.

The caching solutions usually operate as a black-box for simplicity, employing standard cache replacement policies such as Least Recently Used (LRU). In general, every application implements its own caching layer tailored to its specific requirements. In short, these above problems can be seen as basic concern of cache services, in particular, management and replacement. Thus, we require a caching framework, which facilitates the construction process of cache services in a variety of applications. More precisely, such framework should be flexible and scalable to different environments, infrastructures or requirements regarding to architecture of DMS. Although the frameworks of cache services have been studied, most of them are presented with respect to traditional caching mechanism, in particular, page or block cache.

Conventionally, cache utilization like page or block seems to be low, as even one needed record or value in a page/block requires to retrieve and cache the entire page/block, leading to waste valuable space in the cache. Moreover, there is no exploitation of the knowledge

of queries themselves or checking their partly equivalence. Thus, these cache mechanisms can result to a low hit ratio, especially in applications with a chain of queries in order to refine the results such as in log analysis domain.

As an alternative approach to the traditional caches, Semantic Caching (SC) overcomes these issues by exploiting resources in the cache and knowledge contained in the queries. Generally, most of the SC approaches evaluate the queries and reuse their information from logical description, named *semantic*, rather than checking satsifiability of cached data with respect to query condition. To do that, SC has to rewrite the original query if necessary to new relevant sub-queries split in *answerable (probe)* and *non-answerable (remain)* parts of cached data. Nevertheless, the complexity of this procedure, named *query rewriting*, can induce a high overhead in checking equivalence or satisfiability because of its excessive computation. Noticeably, beside the algorithm of query rewriting, the capability of the infrastructure plays an important role to solve the presented issue in SC.

Basically, to maintain the performance of the infrastructure, there are two approaches in terms of scaling-up (vertical scaling) with hardware resources. In particular, upgrading Central Processing Units (CPUs) with multiple-cores technology or using more of them as the first approach. Since CPU has "power wall" limitation (Moore's law), as an alternative approach, replacing or accelerating with high computing specialized hardware, in particular, Field Programmable Gate Arrays (FPGAs) have been proposed. Indeed, FPGAs have been noted to be good candidates for their high parallelism of multi-tasks, reconfigurability, low power consumption, and can be attached to the CPU as an IO device accelerator. Additionally, FPGAs have been accepted gradually in many studies including accelerations of database analytic or even commercial products. Therefore, *FPGAs can be seen as the candidate for two objectives at the same time: first, an acceleration for query rewriting and second, part of query executing with respect to semantic cache concepts.*

In summary, these different state-of-the-art elements, such as cache framework, semantic caching with query rewriting, and FPGA-based database acceleration, seem to be interesting to combine together. Therefore, we are interested in studying *a flexible and scalable framework for cooperation between procedures of SC and accelerators on FPGA*. To conclude, in this dissertation, our contributions is many-fold: 1) We present a modular approach to make SC is flexible, scalable and adaptable with different requirements, environments and infrastructures. 2) We propose a novel approach of cache management in terms of coalescing strategy and replacement policy. 3) We implement a mechanism to handle select-project-join query in SC. 4) We exhibit a cooperative model between SC

and FPGA where query processing is accelerated regarding to query rewriting and part of query execution.

The first contribution is ModulAr Semantic CAching fRAmework (MASCARA) as a cache management system (CMS) in the middleware layer of DMS. Our proposed framework, MASCARA divides and regroups the functionalities, computations and procedures of SC into modules and stages. Thus, the main contribution of this architecture is about the flexibility, scalability and adaptability to different environments, infrastructures and requirements. The experimental results exhibit the performance of SC in different aspects, such as response time, hit ratio and transferred data from storage. The best case shows that MASCARA is up to 3.9 times faster than the baseline (e.g., block cache). In contrast, we analyze a significant decline of response time in MASCARA when the query complexity increases in terms of dimensions, e.g., in the worst case, MASCARA is 2.4 times slower than baseline.

The second contribution is cache management in MASCARA, from coalescing strategy to replacement policy. We revisit the impact of conventional coalescing strategies, such as Always and Never Coalescing. Then, we propose an heuristic solution which strikes a good balance between the two extremes. By this way, the heuristic can increase hit ratio and reduce cache space usage of MASCARA. Howoever, the experimental results show that Coalescing Heuristic is 2.3 times slower than Always Coalescing.

The third contribution is MASCARA with Multi-view processing to handle (inner) join queries. We present an approach, named Multi-view processing, which decomposes an original (inner) join query into (select-project) sub-queries that belong to different joined relations or views. However, this approach can cause a significant execution time of Query Trimming in MASCARA due to the process of multiple generated sub-queries. The experimental results show that performance of MASCARA based on CPU can be reduced significantly. In particular, it runs 1.7 and 3.6 times slower than No-Cache and Block-Cache when the dimension of the query and the number of segments is high.

The fourth contribution is MASCARA-FPGA, a cooperative model to accelerate range query processing. We develop an integration combining MASCARA and FPGA which supports the execution of select-project-join range queries. To achieve this goal, we design the query rewriting of MASCARA and their tasks with respect to FPGA accelerators in bottom-to-top pipeline execution. We also develop the essential DB operators on FPGA to execute the generated sub-queries from query rewriting, such as filter, project and sort-merge-join. By coordinating all of them together, MASCARA-FPGA with single instance

for every accelerator, is able to accelerate on average 6.9 times compared to MASCARA based on CPU. It is worth noting that these accelerations can increase gradually if deploying multiple instances of accelerators on FPGA. Since MASCARA-FPGA can handle the drawbacks of coalescing heuristic for semantic management and multi-view processing for (inner) join query, we revisit their benefits. On the one hand, MASCARA-FPGA with heuristic exhibits an acceleration up to 2.5 times compared to MASCARA-Server with Always Coalescing as the best case on a server. Additionally, heuristic also has the highest hit ratio and a balance space usage in cache compare to the two conventional approaches. On the other hand, overcoming the presented issue of Multi-view processing, MASCARA-FPGA can maintain a high acceleration (e.g., on average 8.6 times for total response time).

Resumé

Ces dernières années, nous avons assisté à un changement de paradigme dans les systèmes d'analyse masses de données. En particulier, comme alternative aux systèmes de gestion relationnelle des bases de données (RDBMS) stockées localement, il y a une émergence de nouveaux systèmes de gestion des données (DMS) où celles-ci sont stockées à distance et distribuées. Ainsi, les ressources de calcul peuvent y accéder de manière indépendante. Compte tenu de la latence relativement élevée dans la communication entre les couches de données et les ressources de calculs, le temps d'accès aux données est accru considérablement. Cependant, la mise en œuvre d'un système de cache peut accroître la réutilisation à grains fins des données et atténuer par conséquent les réexécutions inutiles de requêtes. La technologie de mise en cache pour ces DMS conserve les données fréquemment utilisées par la couche de calcul dans un stockage local et rapide (p. ex., la mémoire principale) dont la taille est limitée accélérant ainsi le traitement des requêtes.

Ces solutions de mise en cache fonctionnent simplement sur la base de politiques de remplacement utilisées classiquement dans les caches standards tels que "moins utilisés récemment", LRU en anglais (*Least Recently Used*). En général, chaque application met en œuvre son propre service de cache adapté à ses propres exigences. La gestion et le remplacement en mémoire peuvent être considérés comme une préoccupation fondamentale et nécessitent la définition d'un environnement de développement qui facilite la construction des services de cache pour une grande variété d'applications. Plus précisément, cet environnement devrait être flexible et évolutif selon les applications, les infrastructures ou les exigences en matière d'architecture des DMS. Bien que les systèmes des services de cache aient été étudiés précédemment, la plupart d'entre eux exploitent les mécanismes de mise en cache traditionnels (cache de pages ou de blocs).

De manière classique, l'efficacité d'utilisation du cache par page ou bloc semble faible, car l'enregistrement d'une valeur nécessite de récupérer et de mettre en cache la page ou le bloc en entier, ce qui entraîne un gaspillage d'espace précieux dans le cache. En outre, il n'y

a pas d'exploitation de la connaissance des requêtes elles-mêmes ou de vérification de leur équivalence partielle. Ainsi, ces mécanismes de cache peuvent conduire à un faible taux de réponse positive (*cache hit*), comme lors d'une succession de requêtes dans l'objectif d'affiner les premiers résultats obtenus.

En tant qu'approche alternative, le Cache Sémantique (SC) surmonte ces problèmes en exploitant à la fois les données dans la mémoire cache et les informations contenues dans les requêtes. Généralement, la plupart des approches SC évaluent les requêtes et réutilisent leurs informations à partir d'une description logique, appelée sémantique, plutôt que de vérifier la compatibilité des données mises en cache par rapport à l'état de la requête. Pour ce faire, un SC doit, si nécessaire, réécrire la requête originale en de nouvelles sous-requêtes pertinentes et basées sur la connaissance des données mises en cache (*probe*) et les données absentes du cache (*remainder*).Néanmoins, la complexité de cette procédure, appelée réécriture de requête, peut induire un coût de calcul élevé dans la vérification de l'équivalence ou de la satisfiabilité. Il est à noter qu'outre l'algorithme de réécriture des requêtes, la capacité de l'infrastructure joue un rôle important pour résoudre le problème présenté dans SC.

Fondamentalement, pour maintenir les performances de l'infrastructure, il y a deux approches d'amélioration possibles au niveau des ressources matérielles. Tout d'abord, il est possible de faire évoluer les processeurs utilisés vers des générations plus récentes et/ou utilisant plusieurs cœurs. Cependant, la loi de Moore étant maintenant largement répandue, il est communément admis que cette solution présente ses limites. Comme approche alternative, l'utilisation de solution matérielles de type co-processeurs commencent à être proposées (par exemple les FPGA). En effet, les FPGA ont été considérés comme de bons candidats grâce à leur capacité à réaliser des calculs à parallélisme élevé, leur reconfigurabilité, leur faible consommation d'énergie et la possibilité de les lier au CPU comme accélérateurs. De plus, les FPGA ont été utilisés progressivement dans de nombreuses études, notamment pour l'accélération de l'analyse de données y compris dans l'utilisation de produits commerciaux. Par conséquent, dans le contexte qui nous intéresse, les FPGA peuvent être considérés comme de bons candidats pour aussi bien permettre une accélération dans la réécriture des requêtes, mais aussi pour accélérer une partie de l'exécution de ces requêtes en supportant les concepts de cache sémantique.

En résumé, la combinaison de ces différents éléments (l'environnement de développement de cache, l'utilisation des concepts de cache sémantique avec réécriture de requêtes et l'accélération de la base de données basée sur FPGA) nous semblent intéressant à ex-

plorer. Par conséquent, nous nous sommes intéressés à présenter un environnement de développement de cache flexible et évolutif pour la coopération entre les procédures de SC et les accélérateurs sur FPGA. Pour conclure, dans cette thèse, nos contributions sont multiples : 1) Nous présentons une approche modulaire pour rendre le SC flexible, évolutif et adaptable avec des exigences, des environnements et des infrastructures différents. 2) Nous proposons une nouvelle approche de la gestion du cache en termes de stratégie de regroupement et de politique de remplacement. 3) Nous mettons en place un mécanisme pour traiter la requête *select-project-join* dans SC. 4) Nous présentons un modèle coopératif entre SC et FPGA où le traitement des requêtes est accéléré en ce qui concerne la réécriture et une partie de l'exécution de celles-ci.

La première contribution, ModulAr Semantic CAching fRAmework (MASCARA), est un système de gestion du cache (CMS) dans la couche intermédiaire de DMS. MASCARA divise et regroupe à la fois les fonctionnalités, les calculs et les procédures de SC en modules. La principale contribution de cette architecture est la flexibilité, l'évolutivité et l'adaptabilité à différents environnements, infrastructures et contextes d'exécution. Les résultats expérimentaux montrent les performances de SC sous différents aspects, tels que le temps de réponse, le taux de réponse et la quantité de données transférées. Les résultats montrent que MASCARA est jusqu'à $3,9$ fois plus rapide que le niveau de référence (c. à d. cache de blocs implémenté sur Spark). En revanche, nous analysons une baisse significative du temps de réponse dans MASCARA lorsque la complexité des requêtes augmente en termes de dimensions. Par exemple, dans le pire des cas, MASCARA est $2,4$ fois plus lent que le niveau de référence.

La deuxième contribution est la gestion du cache dans MASCARA, de la stratégie de regroupement à la politique de remplacement. Nous revisitons l'impact des stratégies de regroupement conventionnelles *Always Coalescing* et itNever Coalescing. Ensuite, nous proposons une solution à base d'heuristique (*Coalescing Heuristic*) qui établit un bon équilibre entre ces deux stratégies. De cette manière, cette solution peut augmenter le taux de réponses positives et réduire l'utilisation de l'espace cache de MASCARA. Cependant, nous constatons que celle ci n'est pas préférable à utiliser avec MASCARA sur processeur en raison de la complexité de la réécriture de la requête. Les résultats expérimentaux montrent que notre solution *Coalescing Heuristic* est $2,3$ fois plus lente que *Always Coalescing*.

La troisième contribution est MASCARA avec traitement multi-vue pour traiter les requêtes *inner-join*. Nous présentons cette approche qui divise les requêtes *inner-join*

en sous-requêtes *select-project* qui appartiennent à différentes relations ou vues jointes. Cependant, cette approche peut causer une augmentation du temps d'exécution significative de la fonction du découpage de requêtes dans MASCARA en raison du nombre de processus de sous-requêtes générées. Les résultats expérimentaux montrent que les performances de MASCARA sur processeur peuvent être réduites de manière significative. En particulier, en comparaison avec des solutions sans cache et avec caches de blocs, MASCARA fonctionne respectivement $1,7$ et $3,6$ fois plus lentement quand la dimension de la requête et le nombre de segments sont élevés.

La quatrième contribution est MASCARA-FPGA, un modèle coopératif pour accélérer le traitement des requêtes d'intervalle. Nous développons une intégration entre MASCARA et FPGA qui permet l'exécution de requêtes *select-project-join*. Pour atteindre cet objectif, nous réécrivons les requêtes MASCARA et leurs implémentations pour une exécution sur cible FPGA.Nous développons également les opérateurs de bases de données essentiels sur FPGA pour exécuter les sous-requêtes générées, tels que *filter*, *project* et *sort-merge-join*. En coordonnant efficacement les sous-requêtes, en utilisant une seule instance par accélérateur, MASCARA-FPGA est capable d'accélérer en moyenne $6,89$ fois les traitements par rapport à MASCARA sur processeur. Il est à noter que ces accélérations peuvent augmenter graduellement si l'on déploie de multiples instances d'accélérateurs sur FPGA. Nous évaluons MASCARA-FPGA dans sa capacité à améliorer les faibles performances de MASCARA sur processeur au niveau des traitements *Coalescing Heuristic* et multi-vue. D'une part, MASCARA-FPGA avec *Coalescing Heuristic* montre une accélération jusqu'à $2,5$ fois supérieur par rapport au meilleur résultat de MASCARA sur processeur avec la stratégie *Always Coalescing*. En outre, l'approche *Coalescing Heuristic* a également le plus haut taux de réponses positives et une utilisation plus équilibrée de l'espace du cache par rapport aux deux approches conventionnelles. D'autre part, nous résolvons le problème présenté du traitement multi-vue. En effet, MASCARA-FPGA peut assurer une accélération efficace, notamment nous obtenons un temps de réponse total $8,6$ fois plus rapide en moyenne comparé à une exécution sans accélérateur.

Table of Contents

List of Abbreviations

Abbreviation	Definition
RDBMS	Relational Database Management System
DMS	Data Management System
SQL	Structured Query Language
CPU	Central Processing Unit
FPGA	Field Programmable Gate Arrays
GPU	Graphical Processing Unit
MPP	Massive Parallel Processing
HLS	High Level Synthesis
SC	Semantic Caching
PQ	Probe Query
RQ	Remainder Query
AC	Always Coalescing
NC	Never Coalescing
CH	Coalescing Heuristic
DNF	Disjunctive Normal Form
CNF	Conjunctive Normal Form

Chapter 1

Introduction

Abstract: In this chapter, regarding to a renewed spike of caching technology for data management systems over a large and scalable resources, we present our motivations and objectives in terms of a semantic caching (SC) framework with acceleration by Field Programmable Gate Arrays (FPGA). In Section 1.1, we first present the problems of existing solutions with respect to caching framework, SC and FPGA-based database. Then, by coordinating all the above elements, we aim to propose a flexible and scalable framework for cooperation between procedures of SC and accelerators on FPGA. To achieve this goal, we particularly focus to find answers according to several dimensions, such as cache-as-service, query rewriting on FPGA and cache management through coalescing strategy. As the end of the chapter, in Section 1.2, we present the research contributions that will be proposed in details later in the remainder of this dissertation.

1.1 Context and Motivation

In recent years, we have witnessed a paradigm shift in analytical database systems over a large and scalable resources. In particular, as an alternative to relational database management systems (RDBMS) with local data storage, there is an emergence of new data management systems (DMS) with remote and distributed data storage with respect to big data [24] and cloud computing [7]. More precisely, such systems can employ a layer or dis-aggregated storage model, where the elastic *compute layer* accesses data on an independently scalable *remote storage*. For example, what can be deployed on compute layer is high performance large scale data processing engines, such as Apache Spark [8]

and Greenplum [60] based on Massive Parallel Processing (MPP) architectures [67, 14]. Interestingly, these engines or frameworks can be expressed by a single node (machine) or cluster of multiples nodes to provide greater processing power. By this way, they improve data sharing and reduce query latency through efficient memory allocation and execution plans. Basically, they can be built on top of a *storage layer* where the instance can be a traditional RDBMS (e.g., PostgreSQL [46]), a distributed file system (Hadoop DFS [85]), or even remote cloud storage (e.g., Amazon S3 [4]). Additionally, they store consistently large amounts of data in different file formats, such as text, images, or videos regardless of architectures. Therefore, a large number of applications in context of big data can be developed based on these new DMSs, such as DNA analyzing [31], satellite data processing [59] or geometric operations [97].

Given the relatively high latency in communication with respect to low bandwidth between these above layers, we consider that caching data at the compute layer now has become more important. Indeed, a cache system can increase fine-grained re-usability of data to mitigate unnecessary query re-executions. For example, to enhance the performance of security monitoring, a tool, which allows to analyze a large number of log files (e.g., HTTP log files of Apache2 server [83]), can be used. Certainly, to handle a large scale of read-only log files with respect to the contents, tools can be built based on the presented data management system. The objective is to help the administrator (e.g., Bob) to quickly detect suspicious behaviors or unauthorized changes. Let's assume that the first query has been done for a long time due to the fact that log files are large and stored distributively or remotely. Bob wants to refine or filter his concerns by a second query. Without caching system, this query is considered as a totally new one to be executed. Obviously, Bob wasted his time although the result could be answered entirely or partly by previous query's content. This problem has become more serious when Bob needs to run a sequence of queries to get the right answer before taking an action with respect to security protocol. *A cache system would reduce query latency and data communication between engine and storage.* In summary, we are witnessing a renewed spike in caching technology for these DMSs where the hot data is kept at the compute layer in fast local storage of limited size to accelerate query processing.

These caching solutions usually operate as a black-box for simplicity, employing standard cache replacement policies such as Least Recently Used (LRU). In general, every application implements its own caching layer tailored to its specific requirements, resulting in a lot of duplication work across systems, reinventing choices such as what to cache,

where to cache, when to cache, and how to cache. In short, these above problems can be seen as basic concern of cache services, in particular, management and replacement. Thus, we require a caching framework, which facilitates the construction process of cache services in a variety of applications. More precisely, such framework should be flexible and scalable to different environments, infrastructures or requirements regarding to architecture of DMS. This approach has been either studied or deployed by several works, such as adaptable cache services (ACS) [28, 26], Amazon Elastic Cache [3], and in-memory cache Redis [76]. To summarize, most of these frameworks are presented to handle the services with respect to traditional caching mechanism, in particular, page or block cache.

Conventionally, cache utilization like page or block seems to be low, as even one needed record or value in a page/block requires to retrieve and cache the entire page/block, leading to waste valuable space in the cache [30]. On the other hand, tuple cache which is not compatible to be applied in big data since its overhead of data comparison. Additionally, all of them use fault based mechanism (i.e., hit and miss) to answer the query. Moreover, there is no exploitation of the knowledge of queries themselves or checking their partly equivalence. Thus, these cache mechanisms can result to a low hit ratio, especially in applications with a chain of queries in order to refine the results as in above example of log analysis tool.

As an alternative approach to the traditional caches, Semantic Caching (SC) overcomes these issues by exploiting resources in the cache and knowledge contained in the queries. It enables effective reasoning (e.g., analysis and processing), delegating part of the computation process to the cache, reducing both data transfers and CPU load on compute layer. This approach was first proposed by [30] and was later extended and improved by a large body of work [90, 34, 78, 52, 54, 38, 57, 80, 29, 27, 94]. Generally, most of the SC approaches evaluate the queries and reuse their information from logical description, named *semantic*, rather than checking satsifiability of cached data with respect to query condition. To do that, SC has to rewrite original query if necessary to new relevant sub-queries split in *answerable (probe)* and *non-answerable (remain)* parts of cached data. Nevertheless, the complexity of this procedure, named *query rewriting*, can induce a high overhead in checking equivalence or satisfiability because of its excessive computation. More precisely, such kind of checking raises nondeterministic polynomial (NP)-complete problem which is considered as a challenge of query rewriting regardless of algebraic or symbolic approach [18, 19, 98]. Some works present different algorithms to handle this NP-Complete problem [43, 42, 79] which could result to a significant reduc-

tion of SC's performance. Noticeably, all of them agreed that the capability of CPU plays an important role to solve the presented issue. Consequently, *performance and overhead of SC for range query processing are evaluated to show that it can work well in a range of applications based on data management systems*, both in high-bandwidth local area network and network-constrained environments [52, 78].

In this dissertation, we focus on the problem of supporting *non-aggregate Select Project Join (SPJ) range queries for SC with multi-dimensions*. A range query is a fundamental operation in database that enables to express a bounded restriction over the fetched records. For instance, the administrator Bob may pull out the records from a range of Internet Protocol (IP) addresses because he thinks that an abnormal event could happen here. This event can be found by a single or multiple of predicates based on the attribute (e.g., name of column) of data set. Thus, he can use the following SQL-like query with three dimensions, in particular, *ip*, *date* and *time*.

```
SELECT *
FROM log
WHERE ip > '192.168.1.1' AND ip < ''192.168.1.255' AND date > '01-01-2020'
     OR ip >= '192.168.1.255' AND time < '12:00:00'
```

To maintain and enhance the performance of SC, basically, there are two approaches in terms of scaling-up (vertical scaling) with hardware resources [6]. In particular, upgrading Central Processing Units (CPUs) with multiple-cores technology or using more of them as the first approach. Since CPU has "power wall" limitation (Moore's law [9]), as an alternative approach, replacing or accelerating with high computing specialized hardware, such as Graphics Processing Units (GPU) [12] and Field Programmable Gate Arrays (FPGAs) have been proposed [36]. Indeed, FPGAs have been noted to be good candidates for their high parallelism of multi-tasks, re-configurability, low power consumption, and can be attached to the CPU as an IO device accelerator [23], [73]. Meanwhile, GPUs are designed specifically to operate in Single Instruction Multiple Data (SIMD) streams fashion. In fact, using GPU has dramatically evolved over the last few decades to have found extensive use in the research surrounding machine learning, AI, deep learning and database analytic [12] thanks to its mature eco-system. However, the emergence of new generation of FPGA development tools in the recent years, in particular, High-Level Synthesis (HLS), such as Catapult HLS [5] and Vitis HLS [53], has raised again the competition between FPGA and GPU [21]. Specifically, data processing using an FPGA was discussed carefully regarding to heterogeneous many-core systems which exhibit performance figures that are

competitive with modern general-purpose CPUs [62]. Additionally, FPGAs have been accepted gradually in many studies including accelerations of database analytic ([81, 86, 95, 71, 91]) or even commercial products, such as Bing search [17] of Microsoft and Intel with a network of 100,000 FPGAs, and EC2 platform of Amazon [2]. Noting that query rewriting bottleneck comes from a large number of intersection and difference computing tasks between logical descriptions of queries themselves, thus, using FPGAs seems to be more preferable thanks to its pipeline execution model for tasks running concurrently and low power consumption. However, we have to admit that FPGA vs GPU performance comparison is always an open question [21]. For this dissertation, *FPGAs can be seen as the candidate for two objectives at the same time: first, an acceleration for query rewriting and second, part of query executing with respect to semantic cache concepts.*

In summary, these different state-of-the-art elements, such as cache framework, semantic caching with query rewriting, and FPGA-based database acceleration, seem to be interesting to combine together. More specifically, they mainly attempt to deal with the problem independently. For example, FPGA-based database systems are proposed without SC and vice versa. Meanwhile, none of the cache-as-service solution with respect to semantic concept at middle-ware layer is considered. Therefore, we are interested in studying *a flexible and scalable framework for cooperation between procedures of SC and accelerators on FPGA*. By this way, we can fill in the gap between few-expressive hardware acceleration and fine-grained knowledge expensive query processing for data management system in the big data context. To conclude, in this dissertation, we particularly focus in finding answers according to the following dimensions:

First, variety of applications based on data management systems often confront the context where their tailored cache (e.g., block caching) at compute layer cannot bring the highest performance due to limitation of conventional cache mechanism. Meanwhile, SC has not been considered to be deployed as a middleware layer which sits between query engine and storage in the system. In particular, SC in middleware layer can be seen as cache management system (CMS) which can provide fast local storage co-located on compute node and talk to (remote) storage to retrieve data as needed. To achieve this goal, *SC needs to be addressed as cache-as-service solution or a framework* as in [28, 26]. By this way, SC could handle change of workload's characteristics, running environment, deploying infrastructure, application's constraints, etc. Our observation shows that most of existing SC contributions can not satisfy the above condition since their architectures are not flexible and scalable [30, 78, 52, 54, 38, 57, 80, 29, 27, 94]. More importantly,

without presenting the functionalities in terms of computing components, they cause many obstacles in scaling and inheriting, such as customization of query equivalence checking, analyzing bottleneck from excessive computation, and integrating new platform to run query rewriting.

Second, in order to bring more benefits for SC framework, cache management is discussed in terms of *coalescing strategy and replacement policy*. In particular, coalescing strategy can affect the response time, hit ratio, and cache space utilization. Most of the works on SC use two conventional approaches, in particular, Always and Never Coalescing, where their trade-off can be seen as a question [30]. Revisiting the strengths and mitigating the drawbacks from these conventional approaches may allow us to propose a novel appropriate solution in cache management. Hence, a study of coalescing strategies in terms of response time, hit ratio and cache space usage should be considered.

Third, processing select-project queries are seen as basic features of SC. Meanwhile, join processing is more complex due to multiple of participated relations. Indeed, although prior SC solutions present in details select-project queries, none of them describes how to handle *join query* with respect to query rewriting and join result managing. A few of works from other broad line of research, *materialized view* [40, 84, 89], present new query rewriting procedure for join and aggregate query at the same time. However, their approach has a high complexity which reduces significantly cache performance. Additionally, their presented algorithms are not implemented and evaluated, especially in terms of tracking and manipulating joined result. In summary, due to the shortfall of such computing capacity, join query processing in SC has not yet been considered. Therefore, it is essential to discuss a mechanism to process join queries regarding to the SC framework.

Fourth, in addition to the need of *accelerating query rewriting*, such kind of SC framework is expected to leverage FPGA's capabilities, for example, low latency accelerators (kernels) for corresponding computing modules. Moreover, we also consider that the *execution of generated sub-queries* as outputs from query rewriting can be boost-up by FPGA-based database (DB) operators, such as *filter, project, sort-merge-join*, etc. Nevertheless, to the best of our best knowledge, most of the presented schemes of FPGA with respect to data management systems, were proposed purely in accelerating in-memory database (DB) or providing specialized DB accelerators [81, 86, 95, 71, 91, 32, 63, 13, 88]. In other words, there is no state-of-the-art work on FPGA which considering the integration and acceleration of SC. Although this issue was mentioned in [25], it is just a abstract vision about two layers architecture which misses important details, such as how

to integrate, what kind of accelerators, or how to manage cache.

1.2 Research contributions

According to the drawbacks of existing schemes as presented in Section 1.1, we thus resume the research goals of this dissertation. We recall that semantic caching (SC) is emergent to apply in new data management system (DMS) to reduce query latency. Thus, we need a SC framework as cache management system (CMS) in middleware layer with respect to the architecture of the DMS. By this way, it can leverage the fast local storage of compute layer meanwhile taking responsibility in communication with (remote) storage layer. Moreover, such framework has to guarantee the flexibility, scalability and adaptability in different application contexts. To overcome the potential bottleneck of conventional SC, there is a need of combining between SC and FPGA through presenting not only query rewriting but also database (DB) accelerators on FPGA. Cache management in terms of coalescing strategies are also revisited to find a more preferable and optimized solution. Moreover, such framework towards FPGA acceleration should enables query rewriting for not only basic (i.e., select-project) but also complex (i.e., join) queries. As a consequence, we present following contributions:

1) ModulAr Semantic CAching fRAmework (MASCARA) as a cache management system in middleware layer of data management system [**47**]. Our proposed framework, MASCARA, which divides and regroups the functionalities, computations and procedures of SC into modules and stages. More precisely, this work can be done by defining relevant templates, data structures, and interfaces. Thus, the main contribution of this architecture is about the flexibility, scalability and adaptability to different environments, infrastructures and requirements. Moreover, within the modular approach, an analysis of bottleneck of SC can be revisited in details before converting into relevant accelerations on FPGA. We evaluate MASCARA in the context of data management expressed by Apache Spark ([8]) and HDFS ([85]) by running workload of select-project queries (i.e., Q6) and data set generated from TPC-H benchmark [1]. The experimental results exhibit the performance of SC in different aspects, such as response time, hit ratio and transferred data from storage. The best case shows that MASCARA is up to 3.9 times faster than the baseline (e.g., block cache). Meanwhile, it can have a hit ratio up to 91% and thus save approximately 95% data transfer from the storage layer when the cache size is large enough with respect to semantic locality of workload.

In contrast, we analyze a significant decline of response time in MASCARA when the query complexity increases in terms of dimensions, e.g., in the worst case MASCARA is 2.4 times slower than baseline. Finally, we confirmed that several heavy computing tasks of query rewriting in MASCARA take advantage to FPGA accelerators.

2) Cache management in MASCARA: from coalescing strategy to replacement policy [65]. We revisit the impact of conventional coalescing strategies, such as Always and Never Coalescing. Then, we propose an heuristic solution which strikes a good balance between the two extremes. In particular, it can decide when to coalesce data regions based on the recency of usage (temporal locality) and percentage of response contribution (spatial locality) that are presented through a new replacement function. By this way, the heuristic can increase hit ratio and reduce cache space usage of MASCARA. Importantly, we explain why the heuristic is not preferable to use with MASCARA based on CPU due to the complexity of query rewriting. We evaluate our solution by using workload of select-project queries (i.e., Q6) and data set based on TPC-H benchmark [1]. The experimental results show that Coalescing Heuristic is 2.3 times slower than Always Coalescing. Consequently, we consider that the benefits of heuristic will become more remarkable when MASCARA is already accelerated by a specialized hardware (e.g., FPGA).

3) MASCARA with Multi-view processing to handle (inner) join queries. We present an approach, named Multi-view processing, which decomposes an original (inner) join query into (select-project) sub-queries that belong to different joined relations or views. In other words, instead of processing a (inner) join query, we process a list of sub-queries and join their results at the end. However, this approach can cause a significant execution time of Query Trimming in MASCARA due to the process of multiple generated sub-queries. We evaluate our solution by using workload of inner join queries (i.e., Q5) and data set based on TPC-H benchmark [1]. The experimental results show that performance of MASCARA based on CPU can be reduced significantly. In particular, it runs 1.7 and 3.6 times slower than No-Cache and Block-Cache when the dimension of the query and the number of segments is high. Consequently, we expect this issue can be overcome if MASCARA is accelerated with a specialized hardware (i.e., FPGA).

4) MASCARA-FPGA: a cooperative model to accelerate range query processing [66]. We develop an integration combining MASCARA and FPGA which supports the execution of select-project-join range queries. To achieve this goal, we design the query rewriting of MASCARA and their tasks with respect to FPGA accelerators in

bottom-to-top pipeline execution. We also develop the essential DB operators on FPGA to execute the generated sub-queries from query rewriting, such as filter, project and sort-merge-join. We organize cache on off-chip memory (i.e., DRAM) of FPGA which supports a reasonable capacity and high bandwidth connection to accelerators. Besides the main components of MASCARA on FPGA, we also provide some modules to bridge the gap between high level services and low level accelerators, such as FPGA-Adaptor and Query Process Controller. By coordinating all of them together, MASCARA-FPGA with single instance for every accelerator, is able to accelerate on average 6.9 times compared to MASCARA based on CPU, with workload of select-project queries (i.e., Q6) and data set from TPC-H [1]. It is worth noting that these accelerations can increase gradually if deploying multiple instances of accelerators on FPGA. Since MASCARA-FPGA can handle both the drawbacks of coalescing heuristic for semantic management and multi-view processing for (inner) join query, we revisit their benefits regarding to MASCARA-FPGA. On the one hand, MASCARA-FPGA with heuristic exhibits an acceleration up to 2.5 times compared to MASCARA-Server with Always Coalescing as the best case on a server. Additionally, heuristic also has the highest hit ratio and a balance space usage in cache compared to the two conventional approaches. On the other hand, overcoming the presented issue of Multi-view processing, MASCARA-FPGA can maintain a high acceleration (e.g., on average 8.6 times for total response time) with workload of inner join queries (i.e., Q5 of TPC-H [1]).

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 gives background knowledge of big data, semantic caching (SC) and FPGA acceleration for serving the understanding of our contributions. Then, it presents the related works of SC and FPGA-based databases acceleration that motivates our study. Chapter 3 describes a ModulAr Semantic CAching fRAmework (MASCARA) where the definitions and functions of each module are explained in detail. It also exhibit a coalescing heuristic with new replacement value function in terms of semantic management. Moreover, it presents the Multi-view processing in order to handle (inner) join queries. Chapter 4 presents a cooperative model MASCARA-FPGA through relevant components and accelerators within pipeline execution model. Lastly, Chapter 5 gives a conclusion in terms of contributions and future works in perspective of query optimization, SC and FPGA acceleration.

Chapter 2

Background and Related Work

Abstract. This chapter aims at giving the principles and related work about semantic caching (SC) and FPGA-based database acceleration in order to improve performance of new data management system (DMS) in large-scale environment. In Section 2.1.1, we explain the advantages and compatibility of DMS when handling the characteristics of big data applications compared to Relational database management system (RDBMS). Next, in Section 2.1.2, we present the fundamentals of query processing, such as type, operation and dimension that are concerned within the scope of this dissertation. We explain In detail the principles of semantic caching (SC) in Section 2.1.3. Later, in Section 2.1.4, we discuss the acceleration possible gain from specialized hardware (i.e., FPGA) with respect to data processing paradigm. Finally, we summarize the related work from SC and FPGA-based database system in Section 2.2.

2.1 Background

2.1.1 Emergence of new data management systems

In this section, we explain the advantages and compatibility of DMS when handling the characteristics of big data applications compared to Relational database management system (RDBMS).

a) Relational database management system (RDMBS)

Generally, a traditional *database* with varied size can be seen as an organized collection of data with fixed format stored in a centralized architecture [45]. To facilitate the interaction or administration of databases, *database management systems (DBMS)* or in short, *database systems*, maintain the connection between end-users, applications and the database itself [45]. It is worth noting that the term "database" is often used to refer to any of DBMS or applications associated with the database. Traditionally, structured query language (SQL) is used in DBMS to manipulate data within database. According to the *data model* which determines the logical structure of a database, we can have two types of DBMS, *relational* (i.e., RDBMS) and *non-relational* (i.e., NoSQL DMBS). For example, PostgreSQL [46] is a RDBMS, meanwhile, MongoDB [49] exhibits NoSQL mechanisms. In this dissertation, our solution is tailored basically for SQL-like queries in RDBMS where records can be stored by row or column oriented [45].

b) Big data emergence

During the last years, big data have been increasingly used both in the public (research laboratories and government agencies) and private sectors. Big data allows to economically extract value from very large *Volumes* of a wide *Variety* of data, by enabling high-*Velocity* capture, discovery and/or analysis [24]. Data Volume means the size of data, Data Velocity means the speed at which new data arrives and Variety means that data is extracted from varied sources and can be either unstructured, semi structured or structured.

Big data includes several characteristics, such as distributed redundant data storage, parallel task processing, scalability, etc [48] that require a paradigm shift in data management. Indeed, traditional RDMBS seems to be not compatible to handle these above characteristics of big data [37]. In detail, many of query engines at compute tier of RDMBS provides only *vertical scalability* which is also known as *scaling-up* a machine by adding more Central Processing Units (CPUs) or memories. Unfortunately, this approach is expensive and the scalability is limited. In other words, it requires to consider also *horizontal scalability* as a complement, which is also known as *scaling-out* by adding machines in computing cluster. Another problem is about the centralized architecture of storage layer in RDBMS works better when the volume of data is low. As a result, within the big data context, distributed redundant data storage plays an important role. Limitations of RDBMS regarding to big data application are discussed in detail in [37].

c) New data management system

Regarding to the problems of RDBMS, we are witnessing an emergence of new DMS with respect to big data [24] and cloud computing [7]. In detail, the novel DMS can work transparently within various kind of databases, data warehouses, big data distributed storage or even a cloud environment in terms of *storage layer*. Interestingly, cloud storage as a part of cloud computing can also provide large scale data storage solution. It can be classified into three categories: *public cloud* as Amazon [4] or Google [41], *private cloud*, and *hybrid cloud*. More details about these categories and cloud computing services can be found in [7]. Our solution is expected to work with either distributed data storage system or cloud storage service.

Meanwhile, for *compute layer*, DMS also exhibits high performance frameworks, in terms of query engine, which execute big data analytic by splitting them across different nodes in a cluster. Most of these engines are based on Massive Parallel Processing (MPP) architecture [67, 14] where a large number of local or distributed CPUs and/or nodes can simultaneously perform a set of coordinated computations in parallel. Thus, they overcome the limitation of traditional RDBMS and satisfy the scalability issue of computing in big data.

As an example, such a DMS can deploy, at compute layer, a large scale data processing engine like Apache Spark [8]. Meanwhile, at storage layer, Hadoop Distributed File System (HDFS) [85] can be used. A combination between Spark and HDFS seems to be appropriate to represent the functionalities of DMS. It is worth noting that the expensive communication between these two instances or layers when processing query can reduce the overall performance of DMS.

2.1.2 Basic concepts in databases

Since new data management system can work on top of (relational) databases, we present the fundamentals of query processing, such as type, operation and dimension that are considered within the scope of this dissertation.

a) Range query

Range query is current operation that retrieves all records where some value is between an upper and a lower boundary. Let's assume that we have two related data sets, *log* and *users*, that represent connections to Apache2 HTTP server and information on

corresponding user, respectively. Figure 2.1 illustrates the relationship between *log* and *users* as well as their main attributes through Unified Modeling Language (UML). Here, a log can contain one or multiple connection from multiple users.

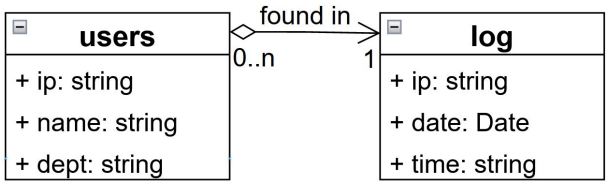


Figure 2.1 – Relationship of *users* and *log* through UML.

An administrator, Bob, wants to find a connection to the server that he thinks may be abnormal. Thus, he can make a range query where he puts some conditions to get the desirable results. One example of such range query (in forms of a SQL query) can be:

```
Q1: SELECT ip, date, time
    FROM log
    WHERE ip >= '134.76.249.10' and ip <= '154.10.10.10'
```

b) Select-project query

The previous range query example consists of two basic operations, *select* and *project*, thus, it can be also seen as a *select-project query*. In detail, the select or selection operator makes it possible to find and filter rows or tuples in a data set that satisfy the query's condition expressed in the "WHERE" clause. Meanwhile, the project or projection operator expressed in the "SELECT" clause, picks the attributes or columns of the data set to be returned.

c) Dimension of query

It is worth noting that the number of attributes which appear in the selection is called *dimension of query*. Generally, since the complexity of query is presented by filtering condition, dimension of query is about the number of attributes in the conditions. For example, the above $Q1$ is a two-dimension (2D) query. If we add *time* in the filtering condition of $Q1$, it will become a three-dimension (3D) query which is considered more complex for processing in semantic caching. To conclude, we focus on non-aggregate range queries since they are one of the most basic operations.

d) Join query

In addition to select-project query, join query is another usual type of query. It includes different operators namely *inner join*, *left outer join*, or *right outer join*, etc. Assuming that Bob not only want to know the suspicious IP addresses related to abnormal connection, but also the owner of these addresses, he can write the following equivalent (inner) join SQL like queries:

```
Q2a: SELECT users.name, users.dept, log.ip, log.date, log.action
     FROM log
     INNER JOIN users ON users.ip = log.ip
     WHERE log.ip = '109.10.10.1' AND log.date = '02-02-2022'
           AND users.dept = 'zone-7'
Q2b: SELECT users.name, users.dept, log.ip, log.date, log.action
     FROM log
     INNER JOIN users ON users.ip = log.ip
           AND log.ip = '109.10.10.1' AND log.date = '02-02-2022'
           AND users.dept = 'zone-7'
Q2c: SELECT users.name, users.dept, log.ip, log.date, log.action
     FROM log
     WHERE users.ip = log.ip
           AND log.ip = '109.10.10.1' AND log.date = '02-02-2022'
           AND users.dept = 'zone-7'
```

Obviously, a join query works with multiple data sets (e.g., *users*, *log*) where there is a relationship between them. For example, *users* connects to *log* through a same attribute, *ip*, which appears in both of them as $users.ip = log.ip$ condition. Basically, to present *explicitly* a join from a logical perspective, a query uses "[TYPE] JOIN ... ON " where [TYPE] is for example with "INNER", "OUTER", "LEFT OUTER", etc with respect to the syntax of the DBMS. Otherwise, join can be *implicit* moving the join condition $users.ip = log.ip$ in the "WHERE" condition. However, this way is not recommended since it reduces the readability of queries. To conclude, in this dissertation, we focus on (inner) join query since it can be converted simply from explicit to implicit format and vice versa with respect to our processing purposes in semantic caching. The other types of join can be handled later by improving the semantic representation for equivalence checking.

2.1.3 Semantic Caching

In this section we describe In detail the principles of semantic caching (SC). We also discuss the advantages and drawbacks of SC. Last, we illustrate the operation of SC through an example of a log analysis tool.

a) Principles

The performance (in response time) of the new DMS (e.g., Spark-HDFS) could be reduced when executing queries that have significant overlaps in different meanings (i.e., redundant execution of certain sub-queries). Indeed, DMS works in context of big data where queries always run over a large scale data sets. Running them without caching re-usable results is not a good idea. For example, around 45% of the queries executed on Microsoft's SCOPE service have computation overlap with other queries [51]. The percentage of overlapping could be increase up to 75% in satellite data processing [59]. Thus, execution of query without re-using the previous answers damages the total response time of system. Recall that this problem comes from costly communication between compute and storage layer. As a result, it increases consumption of computational resources, higher data processing costs, and *unnecessary query execution times*.

To handle this issue, it is necessary to have a cache system which increases data availability at the compute tier in the DMS by answering query rapidly rather than communicating with the storage layer. Generally, most of query engines at compute tier are implemented with tailor page or block cache system to overcome this problem. Moreover, they can integrate a third-party cache-as-service solution as middleware tier in their architecture, such as Redis [76] and Amazon Elastic Cache [3]. Nevertheless, their mechanism is based on either page, block or object that examine the satisfiability of cached answers to be reused with new queries at the *data level*. In other words, they cannot exploit the specified information of queries, named *semantic*, that can be extracted from a query's logical description (i.e., query's condition) at *the semantic level*. Therefore, they are less efficient in answering partial query where the compute layer needs a smaller portion of data from the storage layer rather than entire blocks or pages [30].

Semantic Caching (SC) at the middleware layer seems a relevant alternative to address this issue [54]. According to [30], [54], it can provide better performances than presented conventional approaches (e.g., page or block). In particular, SC achieves a significant load reduction in distributed systems and allows to exploit resources in the cache and

knowledge (semantic) contained in the queries themselves. In short, it enables effective reasoning (analysis and processing), delegating part of the computation process to the cache, reducing both data transfers and CPU load on servers [30]. SC is particularly attractive for different applications in specific domains, such as online analytical processing (OLAP) [34], light weight directory access protocol (LDAP) [20], web queries [57] and heterogeneous systems [39]. Moreover, satellite data processing [59], mobile tracking [77], or log analysis tool [25] where accessing to storage layer is expensive, could be interesting for SC.

In this dissertation, we define SC by the following main properties that are referenced from [30, 38].

— A query cache: SC is a query-based cache which stores query results and its identification (blueprint). In other words, SC could be seen as two linked sub-caches where one is built for the answers (called *data region*), and the other is for their logical description (called *semantic segment*).

— Semantic segment: a special data structure to maintain logical expression of query in terms of semantic segment which points to its corresponding data region answer. Besides logical expression, segment consists of other information, for example, timestamp or LRU value for the replacement policy. This approach makes SC a *logical cache* and distinguishes it from conventional approaches.

— Disjointedness: such characteristic depends on the design choice. According to the design of [78, 30], no data is stored more than once in SC. Hence, redundancy in the cache is reduced. Meanwhile, in [54], SC allows to store duplicated tuples.

— Partial answering (as shown in Figure 2.2): SC reuses *partial matches* of previous query results. Each query which is processed by the SC is split into two disjoint parts. First, the *Probe Query* (*PQ*) is the intersection between the query and segments. It is completely answered by the cache at middleware layer. Second, the *Remainder Query* (*RQ*) which is the difference between the query and the segments, it thus requires to be executed at the compute layer with data coming from the storage layer. Such kind of execution could take a long time due to the communication between these two layers. As a result, the final answer will be combined from the *PQ* and the *RQ*. Such a procedure of finding *PQ* and *RQ* is called *query rewriting*.

Besides its advantages, SC has impediments for its growth and its application in DMS. First, SC can have a heavy execution (i.e., *RQ*) which could take long time, to fullfill the

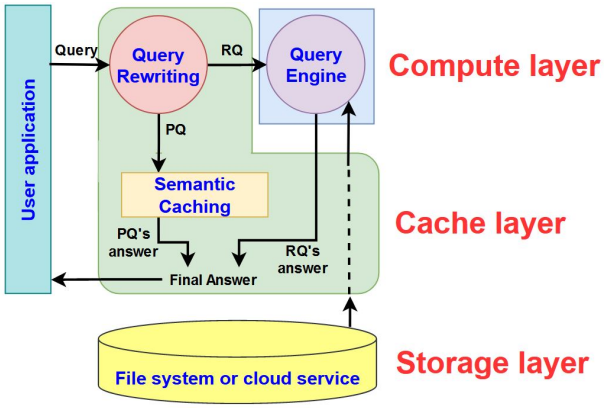


Figure 2.2 – Semantic caching with partial answering.

final answer. Fortunately, the probability of such computation could decrease over time if workload has a relatively high semantic locality [30]. In particular, new coming queries can be totally answered by previous queries since they may consists in a refining procedure of the user. For example, within log analysis tool, assuming that Bob often wants to refine his requests gradually to filter potential IP addresses, the appearance of *RQ* is alleviated.

Second, SC manages non-overlapping data regions with (non-overlapping) associated segments that can help reducing the complexity of query rewriting. Although this is desirable, non-overlapping regions impose a non-negligible overhead [30]. In detail, finding the intersection and difference between queries and segments in query rewriting grows combinatorially with the number of segments and their complexity [43, 42, 79]. Actually, regarding to big data, a cache can hold *several thousands or even millions* of segments that are presented in multi-dimension, the aforementioned problem is expensive to solve.

Third, maintaining and manipulating a large number of segments that have relationship, become complex and cumbersome. Thus, the mechanisms and policies required for managing the cache efficiently (e.g., hit ratio) become challenging. In fact, this problem can be simplified by keeping the segments in the cache independent of each other. However, it can lead to a sequential query rewriting for all of the segments in cache instead of potentially related elements. We doubt that this problem could not be handled well by scaling-up the hardware resources of compute tier (e.g., CPUs, RAMs) since they will soon reach "power wall" limitation [9] when processing a huge number of computing iterations. Furthermore, distributed tracking and manipulating can become more severe when scaling-out in terms of using multiple nodes.

Fourth, SC saves spaces by using dynamic data regions rather than fixed pages or

blocks to store results. In other words, SC only needs receiving exact portion of missing data from the storage layer. Thus, this requires the cache and the storage layer to have a same organization in terms of data unit. However, it is difficult to deploy such kind of agreement from a low-level perspective (e.g,. file system) with respect to the variety of data sources and formats in the storage layer integrated into DMS. To alleviate this problem, a cache can be considered as a flexible and adaptable intermediate interface or middleware layer, which is co-located with compute layer in DMS.

To conclude, with the emergence of specialized hardware in the compute tier in terms of scaling-up, these above problems of SC can be mitigated by advanced parallel architectures, such as Single Instruction Multiple Data (SIMD) of Graphics Processing Unit (GPU), and task parallelism within pipeline of Field Programmable Gate Array (FPGA). More precisely, SC could be deployed as a cache management system (CMS) which is co-allocated on the compute layer and is able to interact with the storage layer. In other words, a key design goal is to make SC sufficiently generic so that it can be plugged into an existing big data system with minimum engineering effort. To achieve this goal, the aim is to serve as the middleware layer between big data systems and distributed storage or even cloud service, exploiting fast local storage in compute nodes to reduce data accesses remotely.

b) Example

To understand well SC, we present in this section an example of the query rewriting procedure. Assuming that we have a log file, named *log*, which has a format such as the one referenced by EDGAR [82]. In particular, *log* consists of several attributes (columns), such as *ip*, *date*, *time*, etc. We have three SQL select-project range queries $Q3$, $Q4$ and $Q5$, to be posed and processed in the following order:

```
Q3: SELECT *                                          Q4: SELECT *
    FROM log                                              FROM log
    WHERE ip > '127.1.0.0' AND ip < '127.1.255.0'         WHERE date < '2020-03-16'
Q5: SELECT *
    FROM log
    WHERE (date > '2020-01-16' AND date < '2020-05-16')
       AND (ip > 127.0.0.0 AND ip < 127.1.255.255)
```

Assuming that $Q3$ and $Q4$ have already been executed and that their answers are now stored in the SC at the middleware layer (as shown in Figure 2.3). Since the segments of $Q3$

and $Q4$ overlap each other, their answers in SC could be maintained in terms of a combined non-rectangle (dotted line) data region. Indeed, SC can use a segment which has the logical description: $Q3 \cup Q4$: $(ip >' 127.1.0.0' \wedge ip <' 127.1.255.0') \vee (date <' 2020-03-16')$. It is worth noting that answers of $Q3$ and $Q4$ in terms of segments could be maintained in other ways. For example, the segment $Q3$ can be left unchanged meanwhile the RQ of $Q5$ represents the new results to be stored in cache [78]. By this way, we have two independent segments, original $Q3$ and RQ_{Q4}.

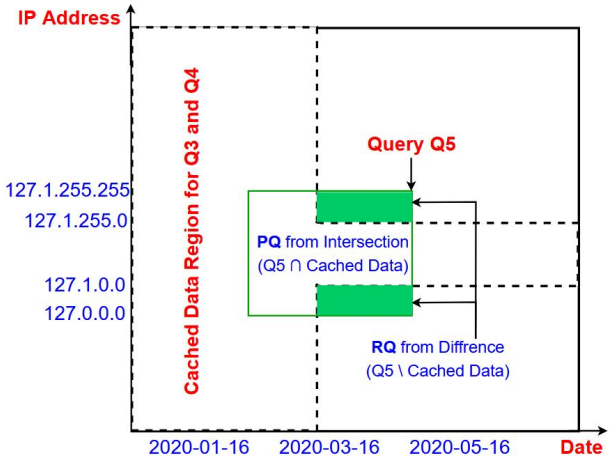


Figure 2.3 – Example of query matching in Semantic Caching

When $Q5$ is posed, the relationship between $Q5$ and $Q3 \cup Q4$ could be identified as the green bounded rectangle. In particular, the SC needs to compute the PQ from the intersection and the difference between $Q5$ and $Q3 \cup Q4$. Then, PQ can be answered rapidly by the compute layer with the cached data indicated by the combined segment of $Q3 \cup Q4$. Meanwhile, the RQ should be slowly executed by retrieving the data from the distribute file system at the storage layer. The final result of $Q5$ is the combination between the answer of PQ and RQ.

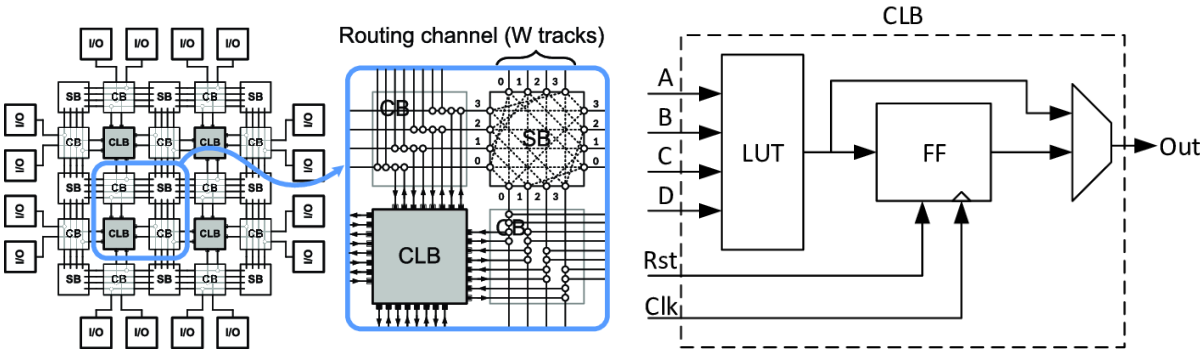
Obviously, in the previous example, we consider a small number of segments (e.g., three segments) where logical descriptions are not complex. Moreover, it consists of only two attributes *ip* and *date* which means that the SC has two-dimension (2D). All together, the complexity of query rewriting in this example is small. Practically, such complexity increases dramatically when processing the difference in the generation of the RQ that involved in generating the RQ can become very intense quickly.

2.1.4 FPGA's acceleration

In this section, we first present the principles of FPGA's architectures. Then, we discuss the advantages of data processing paradigms in CPU, FPGA, and GPU to clarify why FPGA can play an important role in DMS. Last but not least, we introduce the high-level synthesis HLS in developing accelerated solutions on FPGA.

a) Principles

The computational capacity of the Central Processing Unit (CPU) is currently not improving as fast as in the past, or not growing fast enough to handle the significantly growing volume of data with respect to big data. Even though CPU core-count continues to increase in terms of scaling-up, power per core from one technology generation to the next one does not decrease at the same rate and thus the "power wall" limits progress [9]. Unfortunately, this issue remains despite the scaling-out of compute layer in DMS. Furthermore, limitation of CPU's capability can happen similarly with respect to the issues of query rewriting and manipulating the segments in SC. Therefore, we are witnessing a demand of new architectures for not only data processing in DMS but also query rewriting in SC, for example, Field Programmable Gate Arrays (FPGA).



(a) Overview of FPGA architecture. (b) Internal structure of CLB.

Figure 2.4 – Island-style global FPGA architecture. A unit tile consists of Configurable Logic Block (CLB), Connect Block (CB) and Switch Block (SB) [75].

FPGAs have been acknowledged for their high parallelism, re-configurability, and low power consumption [56]. Moreover, they can be attached to the CPU as an IO device to accelerate the analysis in DMS [81, 86, 95, 71, 91]. [75] presented In detail the simplified architecture through the perspective of island-style global which is predominant in commercial FPGA (in Figure 2.4). To begin with, there are identical *unit tiles* arranged in a

rectangular grid interconnected with routing channels of width W. In detail, each of them consists of three types of blocks: *Configurable Logic Block* (CLB), *Connect Block* (CB) and *Switch Block* (SB) (as shown in Figure 2.4a).

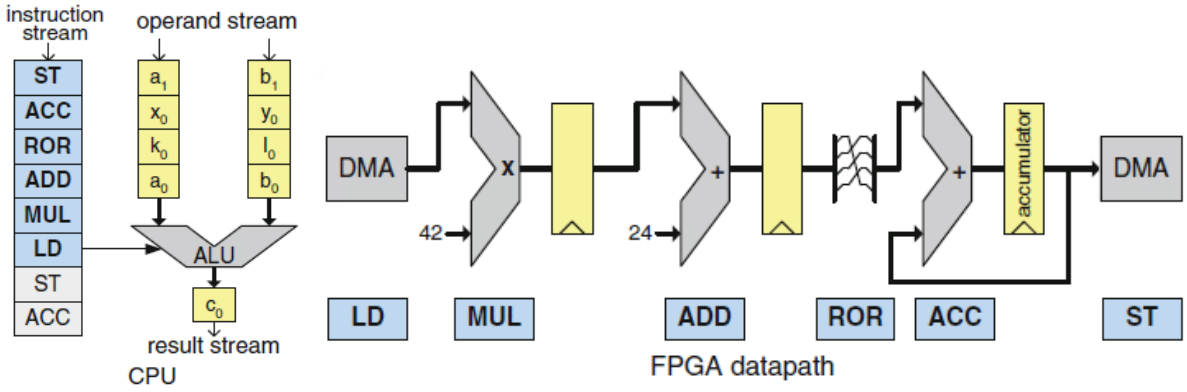
As one of the main component, CLB provides re-configurable logic based on *Look Up Table* (LUT), *Flip-Flop* (FF) and *Multiplexer* (as shown in Figure 2.4b). It is worth noting that these blocks are connected to I/O resources where they can receive analog and digital inputs and return back the output at the end of computations. The advantage of LUT is that it provides a fast way to retrieve the output of a logic operation since possible results are stored and then referenced rather than computed. To save logic states during the logic computation of CLB, FF with a single bit is used to represent two stable states (e.g., 1 or 0). At the end the output of CLB is selected thanks to the Multiplexer. Moreover, a set of *Block RAMs* (BRAM) surrounds the organization of CLBs. Specifically, they can be seen as on-chip memory which works as local and rapid storage for computation of CLBs rather than communicating with an external memory (i.e., off-chip DRAM) of FPGA. However, their capacity is very small compared to the DRAM.

Another block is CB which connects selectively inputs/outputs of CLB to designated routing tracks in a channel through configurable routing multiplexers. As a complement of CB in on-chip programmable interconnect network, SB provides connectivity between horizontal and vertical routing channels, allowing routing tracks to either extend along their current channel or turn a corner to a different channel.

b) Compute paradigm

In this part, we discuss the distinct models, also known as *compute paradigm*, between CPU and FPGA, to clarify why FPGA is considered as a solution for manipulating and managing data in SC (as shown in Figure 2.5).

In Figure 2.5a, CPU uses most popular compute paradigm where it fetches sequentially instructions (from the instruction stream), decodes them, fetches operands (from the operand stream), processes them by an Arithmetic logic unit (ALU) and writes them to a register file or memory. The shortcoming of this model can be: 1) instructions (e.g., fetch, decode, execution) processed in a sequential stream, 2) considerable hardware resources on chip for instruction fetch and decode, 3) power consumption, and 4) I/O throughput for getting the instruction stream. Meanwhile, in Figure 2.5b, FPGA presents a *re-configurable datapath* to process the same instruction stream. Instead of performing different instructions on a single ALU as in the CPU, a chain of dedicated processing



(a) CPU processing. (b) FPGA processing.

Figure 2.5 – Compute paradigm of CPU and FPGA [55].

stages forms a pipeline execution. Thus, in such *datapath paradigm*, there is no instruction stream and the original instruction sequence is decomposed in a chain of simple arithmetic operators. In other words, there is no need for instruction fetch, decode or execute since they are encoded directly into the structure of the datapath. Therefore, FPGA presents better performances than CPU thanks to its highly specialized elements in terms of pipeline model.

Within modern CPU, to fulfill demands for more performance, parallel execution can be applied by distributing the problem to multi cores. More precisely, instead of running sequentially through a piece of code on one core, multiple cores are working side-by-side in this model. Similarly, parallel execution by more processing elements can also be done on FPGA. Additionally, FPGA enables different processing elements to perform different operations (typically in a pipelined fashion) at the same time. In contrast, with multiple CPUs, with a sequence of instructions, it is not easy to parallelize by executing the first half of the instructions on one CPU and the other half on another CPU. Therefore, parallelizing an algorithm on FPGAs can in some cases be easier than multiple core processors [36, 55].

We also acknowledge the strong growth of Graphic Processing Unit (GPU) as a competitor of FPGA in terms of specialized hardware for acceleration. In particular, GPUs use a large amount of *lightweight vector processing elements* that can process multiple input data in forms of vector(s). By this way, sharing the same instruction stream (i.e., same control flow), GPU can process in parallel multiple values. Such paradigm is called *Single Instruction Multiple Data* (SIMD) and can be implemented in terms of *Thread Level Parallelism* (TLP). In short, GPU performs well on data-parallel problems that can

be vectorized and for problems that needs control flow and synchronization with other threads or tasks as in multi-core CPU. In other words, it is still based on control flow paradigm where the instruction needs to be processed similarly to CPU (i.e., fetch, decode, execute). It is worth to remind that FPGA with datapath paradigm can implement any digital circuit that follows architectures of CPU, vector processing, or GPU model.

To conclude, there is not a clear winner between FPGA and GPU acceleration. Depending on the problem and requirements, one or the other is better [55, 21]. However, according to [21], FPGA has shown more efficiency (i.e., performance and energy) for almost all application domains against MPP with respect to big data compared to GPU. Therefore, in this dissertation, with a rising demand for parallelizing algorithms, in particular, query rewriting of SC towards new DMS, it is worth looking into FPGA acceleration [36, 56].

c) High level synthesis

The user-defined logic in FPGA is generally specified using a hardware description language (HDL), mostly VHDL or Verilog. Unlike software programming languages, they require developers to have knowledge on digital electronics design, meaning understanding how the system is structured, how components run in parallel, how to meet the timing requirements, and how to trade off between different resources. Fortunately, with the emergence of FPGAs in database acceleration and else where the development tool chains are improving. In particular, these improvements range from *high-level synthesis* (HLS) tools to domain-specific FPGA generation tools such as query-to-hardware compilers [36]. HLS tools such as Vitis HLS [53] and Altera OpenCL [87] allow software developers to program in languages such as C/C++ but generate hardware circuits automatically. In other words, HLS users write C code meanwhile the interface protocol and micro-architecture are done, generated and managed automatically by the platform. Since our work mainly requires an improvement of query rewriting through the parallelism of complex tasks, it can be done within the software acceleration perspective. Specifically, the HLS approach (i.e., Vitis HLS [53]) seems to be more compatible and preferable with our objective.

Basically, our accelerated solution consists of two distinct components: a software program (on host) and an FPGA binary containing hardware accelerated kernels (i.e., query rewriting of SC, database operators). In detail, the host program in C/C++ runs on a conventional CPU. It can use Application Programming Interfaces (APIs) with respect to a runtime library of the platform to interact with the acceleration kernel in

the FPGA device. Meanwhile, the accelerators (kernels) are written in C/C++ and run within the programmable logic part of the FPGA device. They can be integrated into a software/hardware platform using standard interfaces for interconnection (i.e., AXI).

In fact, there are multiple ways by which the software program can interact with the hardware kernels. We illustrate a simplified communication between kernel and host in Figure 2.6. Generally, the step 1, 3, 5 and 6 allow to read and write results between host program and kernel. We can create corresponding buffers in (logical) global memory which can represent the device memory (i.e,. DRAM of FPGA) or shared memory. Moreover, the host program has to define the events with relevant triggers to enable the operations on kernels (i.e., step 2). Meanwhile, with step 4, kernel performs the required computation, accessing global memory to read or write data, as necessary. Additionally, it can use streaming connections to communicate with other kernels. When the tasks are completed, a notification is created and sent from the kernel to the host (i.e., step 7).

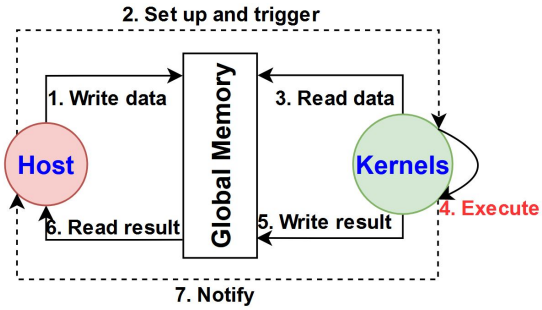


Figure 2.6 – Simplified interaction between host and kernels.

2.2 Related work

2.2.1 Semantic Caching of SQL Queries

SC has first been proposed by S. Dar et al. [30] for simple select-project queries on single relations. In particular, they introduced the basic terminology of SC, including the concept of probe query and remainder query. They compared SC with tuple and page caching to show that SC generally outperforms them. In addition, they studied cache management in terms of replacement policies and coalescing strategies for SC. However, their work did not cope with join queries.

A. Keller and J. Basu [54] presented an abstract SC framework for select-project

and join queries. They also discussed the challenge of maintaining cache currency due to inserts, updates and deletes by presenting a control protocol. Although providing concepts, designs and directions for SC, there was no implementation to evaluate the effectiveness of query processing.

P. Godfrey and J. Gryz [38] introduced a general logical framework for SC in context of heterogeneous databases or data warehouses to optimize query evaluation. In particular, they specified conditions that were relevant with different types of overlaps and subsumptions between queries. In other words, they focused mainly on answering the queries without considering the characteristics of workload and cache management that could impact to performance. Nevertheless, they did not have any kind of evaluation for their proposal.

D. Lee and W. Chu presented CoWeb [57], a SC for web sources. Typically, web sources provided less querying possibilities than traditional database systems. Hence, they introduced a query matching algorithm which finds the best matched query based on the capabilities of the web source. Their work only supported query with conjunctive predicates and it does not allow Join. In addition, they studied semantic locality, i.e., the similarity among queries, and its effect on SC through USAir Flight Schedule dataset [93].

Q. Ren et al. [78] were the first to introduce a formal SC model in which they described segments and query processing with detail implementation. Their SC only allowed select-project queries on single relations. They explained coalescence and decomposition, which are needed to avoid redundant data in SC. In addition, they studied the impact of physical organization in database like indexing and clustering, the query workload, and the network bandwidth on SC through the Wisconsin benchmark [35].

B. Jónsson et al. [52] revisited SC for select-project queries over single relation. They presented a detailed performance analysis of SC using various workloads from simple to complex within Wisconsin benchmark [35]. For simple workloads, their performance showed that SC produced low overhead, was insensitive to clustering, unburdens the network, and can answer queries without contacting the server. For more complex workloads, they showed that SC strained the server due to the complex remainder queries that required more effort. However, they proved that SC uses the network efficiently and successfully reduced query response times. Therefore, they concluded that SC context can be applied to various workloads in a wide range of applications.

N. Ryeng et al. [80] applied SC as site autonomous caching in context of distributed database system. Thereby, the caches of nodes in their distributed database system were

not limited to base tables, but can contain intermediate results. Their work supported all types of queries, included aggregate and join, but the corresponding query execution was not mentioned. They also studied different Least Recently Used (LRU)-based replacement policies, for example, LRU with cost or height. They evaluated their system with the TPC-H benchmark [1] to show the high hit ratio of SC as well as an improvement in execution time. Thus, they concluded that SC in distributed database systems enabled scaling the system without excessive network traffic.

d'Orazio et al. [29] designed Dual Cache to improve select-project query evaluation over data sources distributed across a grid. Dual Cache managed a pair composed of a query cache and an object cache. In particular, the query cache managed query results and object cache is linked to the query cache thanks to the identifiers. Their work was evaluated by using a biological database of protein sequences from Swiss-Prot1 [74]. Thus, they showed that they can maximize advantages of querying caching which were the reduction of both data transfers and query computation. Later, they proposed to extend their work [27], to adapt it to pervasive grids. This was a collaborative cache system based on a lightweight mobile dual cache for Mobile Station (MS) and a proxy dual cache for Mobile Support Station (MSS).

Vancea et al. [94] presented CoopSC that supports n-dimensions range select-project queries for enhancement of the performance of not only read-intensive query workloads but also the update queries. They claimed that CoopSC was the first one to handle the generic n-dimensions range selections. Moreover, they presented an approach of combining multiple entries to answer a given query in distributed environment thanks to the Distribute Rewriting component. Last, they validated the proposal by using the Wisconsin benchmark [35] which showed their gains in response time and hit ratio regarding the size of cache.

Recenlty, d'Orazio et al. [25] presented a vision of deploying SC in the middleware layer of DMS to handle a large number of HTTP log files. Specifically, their novel idea is the integration of SC and FPGA acceleration. Thus, they described the potentials as well as the challenges of implementing SC through a proposed multi-layer architecture which consists of two sides: server and FPGA. After that, Maghzaoui et al. [61] with the preliminary results showed the feasibility of implementing cache on FPGA. In particular, their prototype with a small (conventional) cache on FPGA can accelerate the response time of simple queries (i.e., with two dimensions). Nevertheless, both of them [25, 61] did not present enough the principles and architecture of SC with FPGA acceleration.

Besides SC in RDBMS for SQL like query, there are other relevant contributions with SC where they focus mainly on the query rewriting procedure [44, 40, 84, 89]. In detail, they studied answering queries using materialized views and/or analyze the NP-complete problem. However, several reasons make them difficult to use in SC. For example, they do not support partial answering or they only consider conjunctive queries. We also have other non-SQL applications with SC, such as XML databases [15, 58, 92] and web queries [16, 10, 72], that have particular characteristics and may not be compatible to apply to SQL query-based database.

2.2.2 FPGA based database system

FPGAs have been integrated with database systems in various ways [36]. Based on interconnection, FPGA can be considered to be used conventionally as *IO accelerator* where task are sent from the CPU to the FPGA via the host memory and then the device memory (as shown in Figure 2.7a). Meanwhile, recent technology allows FPGA to act as a *co-processor* where communication between CPU and FPGA is done through a shared memory (as shown in Figure 2.7b). Furthermore, based on functions, we can use FPGA in RDBMS as *framework* and *specialized accelerators*. In this dissertation, we present a FPGA-based database system with respect to framework and accelerators category.

(a) FPGA as IO accelerator

(b) FPGA as co-processor

Figure 2.7 – Integration of FPGA into database systems with CPU

The RDBMS frameworks include solutions that provide a software and hardware stack for accelerating user-defined database operations. Woods et al. [95, 96] presented *Ibex*, an intelligent database storage framework for MySQL [70]. In particular, they provided a limited set of query processing operations to work directly with data inside the Solid State Drive (SSD). They deployed FPGA into the data path between the data source and the host system.

Owaida et al. [71] provided a hardware-software acceleration framework, called *Cen-*

taur, which consisted of two parts, the hardware part called the "FThreads manager" and the software part called the "Application interface". In particular, the Interface supported API calls in C++ through the User Defined Function (UDF) functionality of MonetDB [50]. Meanwhile, the FThreads manager was responsible for managing the different user-defined hardware in terms of resource utilization, memory accesses, and also pipelining. By this way, Centaur framework facilitated the development of applications to software and hardware perspective where tasks can be optimized by the experts.

After that, Sidler et al. [86] presented *DoppioDB* as an extension of Centaur framework. This work can be seen as a demonstrated system from academia for deploying FPGA to act as co-processor with CPU. In DoppioDB, FPGA can access the host memory directly, and the communication with CPU was done through shared memory. They proved that DoppioDB with FPGA can execute SQL queries for an analytical relational database with over three times speedup when compared to the baseline.

Another framework, called *AxleDB*, that was presented by Salami et al. [81]. AxleDB is a fast query processing FPGA-based framework in which they provide a large subset of SQL queries that are supported by specialized accelerators, such as *filter, arithmetic, aggregate, group by, table join, or sort*. They evaluated AxleDB with five decision support queries from the TPC-H benchmark [1].

Sukhwani et al. [91] presented an FPGA-based query processing engine which was attached to a DBMS via PCIe-3. The supported functionalities are *filter, join and sort*. To increase the throughput, this engine was equipped with a data compression capability. In particular, the data was compressed by the host and decompressed on FPGA as the first step in their query processing pipeline. As a result, processed queries were sent back in decompressed form.

Dennl et al. [32, 33] first presented concepts for on-the-fly hardware acceleration of SQL queries. Then, they proposed a query processing platform that leveraged the partial dynamic reconfiguration capability of FPGAs to better fit the requirements of each query on-the-fly. Their work supported for column and row oriented data store. Moreover, their library consisted of query primitives, such as *filter, aggregation, hash join, and sort*.

Regarding to specialized accelerators, we reviewed a series of database operators on FPGA in terms of accelerators. In general, most of them were inspired from SQL queries since it was almost an universal language for expressing database operations. Database operator accelerators can be dispatched into one of the following categories: *sort, select, join, string matching, filter and arithmetic*. Among them, *join* and *sort* have received a lot

of attentions from the database community due to their impact on query processing. To begin with, Mueller et al. [63] proposed Glacier, the query to hardware compiler, which can compose a digital circuit using various implemented operators. It took a continuous query from data streams and generated a corresponding file written in VHDL that was then translated to a FPGA configuration. After that, they studied in detail the performance of sorting networks on FPGA, such as *odd-even sort and bitonic sort*. [64], but their work did not cover join operator. Casper et al. [13] created a full *equi-join with merge sort* implementation in FPGA for minimizing the bandwidth bottleneck They showed that this implementation was memory-bound, meaning that with advances in memory system technology it had the potential to further exceed the performance of a CPU-only solution. Srivastava et al. [88] further optimized *bitonic sort* for accelerating big data applications. They showed that the last merge operations of merge-sort were a bottleneck for large-scale sorting and replaced them with a heavily-pipelined bitonic network for higher throughput.

To summarize, in Section 2.2.1, SC contributions in the middleware layer of DMS are limited to few solutions, such as [38, 29, 94, 25]. Nevertheless, they are conducted without considering the expansion and inheritance of SC functionalities in case of changing the environment, requirements or infrastructure regarding to the application context. Another point is that most of them did not present an integration of specialized hardware (e.g., FPGA) to handle SC main bottleneck (i.e., query rewriting for multi-dimesion). Unfortunately, FPGA-based databases or accelerations are presented in Section 2.2.2 without considering SC. Consequently, this dissertation expect to renew the interest of SC towards FPGA acceleration as a cache management system (CMS).

2.3 Conclusion

This chapter mainly focuses on background knowledge and related work. More specifically, we present principles of DMS, SC and FPGA-based database acceleration regarding to big data applications. Moreover, we illustrate how SC works through an example of log analysis tool and the benefits of FPGA data processing. We then consider the related work of SC and FPGA-based database acceleration in DMS and their drawbacks. Indeed, to the best of our knowledge, none of the existing schemes consider an integration between SC and FPGA as a framework at the middleware layer of DMS to reduce unnecessary query re-execution. Thus, these drawbacks mainly lead to our contributions that will be presented in the next chapters.

Chapter 3

MASCARA

Abstract: Regarding to the emergence of big data, a large number of applications can be developed based on DMS. To maintain the performance, caching can be a solution when the DMS must execute multiple queries with significant overlaps. In particular, semantic caching (SC) as a cache management system (CMS) is promising to overcome the limitations of more traditional approaches (page and block cache) to answer partial queries. Nevertheless, most of SC existing solutions do not discuss the construction of SC in terms of cache-as-services within the middleware layer of DMS. Moreover, these contributions do not consider the expansion and inheritance of SC's functionalities to the change of running components, system requirements, environments or infrastructures. Therefore, in this chapter, to address such issue, we present ModulAr Semantic CAching fRAmework (MASCARA) as a middleware layer which is co-located with the compute layer in DMS. The key idea behind MASCARA is to divide and regroup the functionalities, computations and procedures of SC into modules and stages. As a result, we increase the flexibility, scalability and adaptability of using SC. Based on MASCARA, we present a coalescing heuristic and new replacement value function in terms of cache management. This approach strike a balance between conventional strategies, such as Always and Never Coalescing, to optimize semantic management of cache. Then, we study the compatibility of such approach in MASCARA with respect to CPU's capability. Moreover, to enable query rewriting of join queries, we introduce the Multi-view concept which enable to run a list of sub-queries from multiple relations rather than the original join query. However, the issue of this approach is a significant increase of query rewriting. Finally, we exhibit experimental results to show the performances and the bottlenecks of MASCARA.

3.1 Introduction

Regarding the emergence of big data, a large number of applications can be developed based on new data management systems (DMS), such as DNA analyzing [31], satellite data processing [59] or geometric operations [97]. Most of them can rely on an two-layer architecture where the elastic *compute layer* accesses persisting data on an independently scalable (remote) *storage layer*. On the one hand, the storage layer of the DMS can be expressed in terms of different databases, data warehouses, distributed file systems and even cloud storage services. On the other hand, for the *compute layer*, DMS exhibits high performance frameworks, in terms of query engine, which can execute big data analytics by splitting tasks across different nodes in a cluster. Such a kind of DMS can be deploy, at the compute layer, a large scale data processing engine like Apache Spark [8] on top of the storage layer, in particular, Hadoop Distributed File Systems (HDFS) [85].

However, the performance (response time) of the new DMS (e.g., Spark-HDFS) could be reduced dramatically when executing queries that have significant overlaps (i.e., redundant execution of certain sub-queries). More precisely, this problem comes from the costly communication between the compute and the (remote) storage layer. For example, around 45% of the queries executed on Microsoft's SCOPE service have an overlap with the other queries [51]. Thus, executing of query without reusing previous answers degrades the overall performance of the system. Moreover, it also increases the consumption of computational resources, data processing costs, and *unnecessary query execution times*.

To handle this issue, it is necessary to have a cache system which increases data availability at the compute tier in the DMS by answering queries rapidly rather than communicating with the storage layer. Generally, most of query engines at the compute layer are implemented with traditional cache systems (e.g., page or block) that examine the satisfiability of the query to reuse cached answers at the *data level*. In other words, they cannot exploit the specified information of queries, named *semantic*, that can be extracted from query's logical description (i.e., query's condition) at the *semantic level*. Therefore, they are less efficient in *partially answering query* when the compute layer only needs a new small portion of data from the storage layer rather than an entire block or page [30].

To overcome this problem, Semantic Caching (SC) can be seen as a candidate to be deployed as a *cache management system* (CMS) between the compute and the storage layer

[54, 30]. Indeed, SC achieves a significant workload reduction in distributed systems and allows to exploit resources in the cache and knowledge (semantic) contained in the queries themselves. In short, it enables effective reasoning (analysis and processing), delegating part of the computation process to the cache, reducing both data transfers and CPU load on servers [30]. Nonetheless, most of the existing works [30, 90, 34, 78, 52, 54, 38, 57, 80] do not discuss the construction of SC in terms of CMS or cache-as-services solution. Indeed, these approaches focus mainly on building a physical cache which depends heavily on the agreement on the same data source organization and format between the compute and the storage layer of DMS. However, such kind of solution becomes a challenge since DMS can change the running components, system requirements, environments or infrastructures regarding to the application context. Therefore, it is essential to provide a SC framework in terms of CMS or cache-as-services solution of DMS. Few proposals, such as [38, 29, 94, 25] discussed this idea but they do not consider the expansion and inheritance of SC's functionalities with respect to flexibility and scalability.

In this chapter, first, we propose a ModulAr Semantic CAching fRAmework (MASCARA) at the middleware layer which is co-located with the compute layer in the DMS. We present MASCARA through the definitions of SC that are referenced from studies of [30, 78]. Our choice is motivated by the fact that their model in terms of definitions and theorems have been proved formally within algebraic approach. More important, the key idea of MASCARA is to divide and regroup the functionalities, computations and procedures of SC into *modules* and *stages*. This approach has been applied in some academic contributions [28, 26] or third-party cache services (e.g., Redis [76], Amazon Elastic Cache [3]) without considering the concepts and principles of SC. In contrast, MASCARA presents templates, data structures, and interfaces to facilitate the customization (e.g., add, remove) and inheritance of computing modules that are based on SC. By this way, we increase the flexibility, scalability and adaptability of using SC with MASCARA with respect to different requirements, environments and infrastructures. Moreover, another benefit of modules and stages is that they can be analyzed in details regarding to their performances. In other words, we can identify excessive computing modules (e.g., query rewriting), that can turn into bottleneck of MASCARA.

Second, most of the presented existing SC use one of two conventional coalescing approaches. In particular, Always and Never Coalescing, which can affect the response time, hit ratio, and cache space utilization. We exploit the strengths and mitigate the drawbacks of above straightforward approaches to propose a novel balanced solution in

cache management, named *coalescing heuristic*. In details, it can decide to coalesce or not data regions based on the recency of usage (temporal locality) and the percentage of a response contribution (spatial locality) that are presented through a new replacement function. Unfortunately, the performance (in response time) of this Coalescing Heuristic can be less impressive compared to Always Coalescing. Thus, we doubt that coalescing heuristic may not be compatible with MASCARA based on CPU.

Third, although prior SC solutions present in details select-project queries, none of them describes how to handle *join query* with respect to query rewriting and join result managing. In fact, since they do not want to increase the complexity of query rewriting, they ignore the solutions proposed from other broad line of research, called *materialized view* [40, 84, 89]. Nevertheless, the presented solutions are not implemented and evaluated, especially in terms of tracking and manipulating joined results. We thus present an approach for query rewriting of select-project-join in MASCARA, named *Multi-view processing*, which brakes down an original (inner) join query into (select-project) sub-queries that belong to different joined relations or views. In other words, instead of processing a (inner) join query, we process a list of sub-queries and join their results at the end. Additionally, query rewriting of MASCARA is expressed as computing modules, thus, processing for join query is equivalent to call Query Trimming multiple times or run them in parallel. Although enabling query rewriting for join queries, MASCARA can induce some overhead due to handling many generated sub-queries from each relation in during the query rewriting. Hence, it can make the bottleneck become worse and reduce significantly the performance of MASCARA.

The contributions of MASCARA in this chapter can be summarized as follows:

1. We present MASCARA with stages and modules that represent the functionalities in SC.
2. We propose a coalescing heuristic and a new replacement value function in terms of semantic management in MASCARA.
3. We provide the Multi-view concept to enable query rewriting of select-project-join queries in MASCARA.
4. We extensively evaluate the performance of MASCARA with respect to the response time, the hit ratio and the transfer data (from the storage layer) within generated data set of the TPC-H benchmark [1].

The chapter is structured as follows. In Section 3.2, we give the basic definitions of semantic caching. Next, we describe in details modules and stages of MASCARA in

Section 3.3. In particular, we first introduce the architecture and principles of MASCARA. Then, we explain carefully the responsibility of each stages and modules, especially the query rewriting procedure in Section 3.3.3. Regarding to conventional coalescing strategies of semantic management presented in Section 3.3.4, we then propose a heuristic with a new replacement value function in Section 3.3.5. Later, we present the Multi-view concept for query rewriting of select-project-join queries in MASCARA in Section 3.3.6. We exhibit experimental results to show the performances and the bottlenecks of MASCARA with respect to our contributions in Section 3.4. Finally, in Section 3.5, we summarize the benefits and limitations of MASCARA in order to raise the challenge of acceleration with FPGA.

3.2 Basic definitions

Since the core of MASCARA is semantic caching (SC), it is essential to provide basic definitions with respect to semantic segments that are referenced in [30, 78]. Our choice is motivated by the fact that their model in terms of definitions and theorems have been proved formally within algebraic approach. Regarding to the presented properties of SC in Section a), SC in MASCARA consists of two linked sub-caches, one is for answers (i.e., records) of query and the other is for logical expressions of queries.

Suppose that we have a relational database D consisting of a number of relations $R_1, R_2, ..., R_n$, for example, $D = R_i, 1 \leq i \leq n$. An instance of a relation R_i is a set of tuples or records, in which each tuple has the same number of Attributes A or Columns as defined in the relation schema of D. Then we have A_{R_i} that stands for the attribute set defined by the schema of R_i and $A = \cup A_{R_i}$ which represents the attribute set of the whole database D. We give the following definition of atomic predicates, through which a SC is constructed:

Definition 3.2.1. *In the database D with its attribute set $A = \cup A_{R_i}$, an **Atomic Predicate** AP is represented in forms of **a op v** where $a \in A$, op can be $\{\leq, <, =, >, \geq, \neq\}$ and v is a constant value in a specific domain.*

It is worth to note that the operator $\neq$ brings much more complexity in reasoning, transforming and rewriting over the predicates AP. In fact, the problem could become NP-complete in the domain of integers [79, 43, 42].

Basically, SC allows to process and store the selection condition or logical expression of a query (i.e., "WHERE" clause with AND and OR operators) in terms of *disjunction of*

conjunctions of atomic predicates AP. In other words, the presentation of such condition is called Disjunctive Normal Form (DNF) in which atomic predicates AP_i are managed in Conjunctive Normal Form (CNF). Thus, we give the definition of DNF as follows:

Definition 3.2.2. *In the database D with its attribute set $A = \cup A_{R_i}$, a logical expression of query is presented by $DNF = CNF_1 \vee CNF_2 \vee ... \vee CNF_n$ where each $CNF_i = AP_1 \wedge AP_2 \wedge ... \wedge PA_n$.*

We illustrate this definition through the following SQL like query:

```
Q1: SELECT ip, date, time
    FROM log
    WHERE ip >= '134.76.249.10' AND ip <= '154.10.10.10'
          OR date = '2020-02-05'
```

As it can be seen, the logical expression of the query is transformed into $DNF = CNF_1 \vee CNF_2$ in which $CNF_1 = AP_1 \wedge AP_2$ and $CNF_2 = AP_3$. The OR and AND operators are represented in forms of $\vee$ and $\wedge$ respectively with regards to the relational algebra. In details, AP_1 is $ip >=' 134.76.249.10'$, AP_2 is $ip <=' 154.10.10.10'$, and AP_3 is $date =' 2020-02-05'$.

Unlike an ordinary cache, SC has to store the logical expression of query in terms of a DNF into a specialized data structure, named *semantic segment* S [30]. Such kind of structure is original, decomposed, or coalesced with the query answer. Moreover, every segment points to its actual content, called *data region*, which is the result of query in terms of tuples or records. The definition of semantic segment is described as follows:

Definition 3.2.3. *In the database D with its attribute set $A = \cup A_{R_i}$, a semantic segment S is represented in forms of $< S_R, S_A, S_{DNF}, S_D >$ where relations is $S_R \in D$, attributes of relation is $S_A \subseteq A_{S_R}$, DNF expression is S_{DNF}, and pointed data region S_D.*

Assuming that we already get the answer for query $Q1$. Thus, a segment is formed with the following elements: $S_R = log$, $S_A = ip, date, time$, $S_{DNF} = CNF_1, CNF_2$, and S_D is the pointer of corresponding $Q1$'s answer set (i.e., collection of satisfied tuples).

Since the segment represents for the query answer, we can qualify the semantic information of query, named *query segment* Q, in the same way as we specify segment. However before queries get answered, corresponding answer of query segment is empty, in particular, $Q_D = \varnothing$). Therefore, we formally define a query segment just as we did for semantic segment as follows:

Definition 3.2.4. *In the database D with its attribute set $A = \cup A_{R_i}$, a query segment Q is represented in forms of $< Q_R, Q_A, Q_{DNF}, Q_D >$.*

To reduce space overhead, the cached segments S do not overlap with each other since it can help reduce the complexity of query rewriting [78, 52]. Thus, the concept of disjointed segments are defined as follows:

Definition 3.2.5. *Semantic segment $S_i = S_{i_R}, S_{i_A}, S_{i_{DNF}}, S_{i_D}$ and $S_j = S_{j_R}, S_{j_A}, S_{j_{DNF}}, S_{j_D}$ with $i \neq j$ are said to be disjointed if and only if*

1. *$S_{i_A} \cap S_{j_A} = \varnothing$ where $\cap$ represents for intersection of two sets.*
   OR
2. *$S_{i_{DNF}} \wedge S_{j_{DNF}}$ is unsatisfiable which implies that there is no relevance between $S_{i_{DNF}}$ and $S_{j_{DNF}}$.*

As a result, we can formally define a Semantic Caching SC a set of disjointed segments S as follows:

Definition 3.2.6. *Semantic Cache SC = is a list of S where $\forall i, j : S_i \in SC \wedge S_j \in SC \wedge i \neq j \Rightarrow S_i$ and S_j are disjointed.*

3.3 MASCARA

3.3.1 Principles

This section describes MASCARA, a ModulAr Semantic CAching fRAmework where the main goal is to divide and regroup the functionalities, computations and procedures into specialized *modules* or *stages*. More precisely, MASCARA presents relevant templates, data structures, and interfaces to facilitate the customization (e.g., add, remove) and inheritance of computing modules that are based on SC. By this way, we increase the flexibility, scalability and adaptability of using SC in MASCARA with respect to different requirements, environments and infrastructures. For instance, the original module of *date comparison* supporting for attribute *date* in data set *log* [82] can be converted and extended to work with *pos* (position) in satellite data processing [59].

As depicted in Figure 3.1, MASCARA consists of four main stages: *Query Broking*, *Query Trimming*, *Semantic Management*, and *Result Refining*. To begin with, when user makes a new query, this query is received, transformed and materialized by Query Broking.

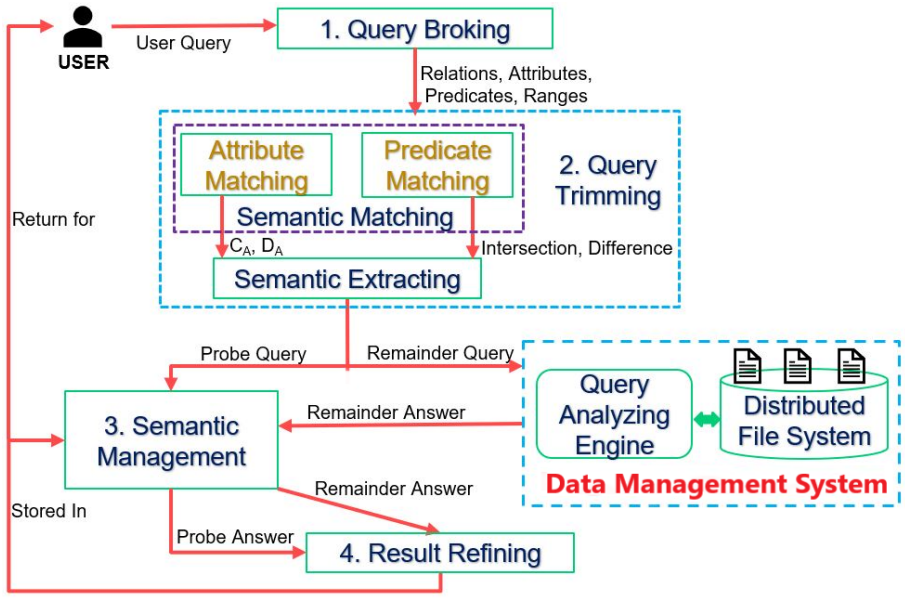


Figure 3.1 – As middleware layer in DMS, MASCARA architecture consists of stages and computing modules.

More precisely, this stage will construct the query segment $< Q_R, Q_A, Q_{DNF}, Q_D >$ based on query extracted information from query, such as projected attributes and logical expression.

Then, this query segment Q is forwarded to Query Trimming, where the query rewriting procedure is implemented. In particular, this Q will be matched against each semantic segment S that are stored in the semantic caching SC. In order to check the satisfiability and implication between Q and S, a sub-stage *Semantic Matching* is proposed, that consists of two modules: *Attribute Matching* and *Predicate Matching*. In particular, Attribute Matching examines the relationship of Q_A, S_{i_A}, meanwhile, Predicate Matching checks the equivalence of $Q_{DNF}, S_{i_{DNF}}$. Since we have to compute the equivalence of each atomic predicate AP in Q_{DNF} and S_{DNF}, it is meaningful to use the term Predicate Matching instead of DNF Matching. Note that checking two perspectives, attribute and predicate, is mandatory to identify that query Q is answerable by S and allows generating new specialized sub-queries in the next sub-stage (i.e., Semantic Extracting).

The outputs of these presented matching modules, such as Common Attributes or Intersection of DNFs, are used to generate the logical expressions of the probe query PQ and the remainder query RQ in the sub-stage Semantic Extracting. RQ, which cannot be answered by the SC of MASCARA, will be handled by the compute layer. Since such

kind of computation requires data transferred from the storage layer, it can consume a lot of time that depends on the size of data set. Meanwhile, PQ is executed rapidly by MASCARA with cached data coming from SC. Generally, SC can be organized either on main memory (i.e., RAM) or disk storage co-located with the compute layer. In this dissertation, we focus on in-memory SC to optimize the benefits of MASCARA which can be brought as middleware layer in DMS.

The results from both PQ and RQ are combined together in Result Refining stage and sent back to user. If there are multiple PQs and RQs, this stage is responsible to track and merge their result in-order or out-of-order in case of matching described by Query Trimming.

Besides the presented computing stages, MASCARA packages the functionalities of cache management on SC, such as coalescing and replacing segment S and its corresponding data region DR, into Semantic Management stage. In particular, it decides to keep, remove or replace the cached data in SC with respect to the applied strategies and policies. Finding appropriate coalescing strategies and replacement policies can improve cache performance in terms of response time, hit ratio and cache space usage.

3.3.2 Query Broking

The main objective of this stage is to transform and materialize a SQL like query to the corresponding elements of query segment $< Q_R, Q_A, Q_{DNF}, Q_D >$. Most of the elements can be extracted straightforwardly from the "SELECT" and "WHERE" clause of query. For example, Q_R receives the relations and Q_A receives the attributes that the query works on. Meanwhile, the element Q_D is empty since answer of query has not been found yet. The procedure to find the value of the last element, Q_{DNF}, is considered as the main functionality of the Query Broking.

In RDBMS, the logical expressions of query are conventionally stored as an *Abstract Syntax Tree (AST)*. Processing on a large number of arbitrarily nested expressions (with AND and OR) can become a challenge later in Query Trimming stage. Therefore, it is essential to transform and maintain the logical expression of query within the Disjunctive Normal Form (DNF) instead of a complex AST.

The construction of the DNF follows two steps. First, from AST, all negations are pushed as far as possible into the tree which results in *Negation Normal Form (NNF)* by using De-Morgan rules. After receiving the NNF, Query Broking can distribute conjunctions over disjunctions. Then, the distributive law pushes OR(s) higher up in the tree

which results in the DNF. To illustrate such procedure, we use the following SQL like query $Q2$. Details of the transformation is depicted in Figure 3.2.

```
Q2: SELECT ip, date, time
    FROM log
    WHERE ip >= '134.76.249.10'
          AND (date > '2020-02-05' OR time > '10:00:00')
```

Based on the NNF of this query, Query Broking transforms from $and(ip, or(date, time))$ to $or(and(ip, date), and(ip, time))$. As a result, the DNF tree could be generated with OR as the root. Although this algorithm within Query Broking can create 2^n leaves in theory, this issue does not become the bottleneck of MASCARA.

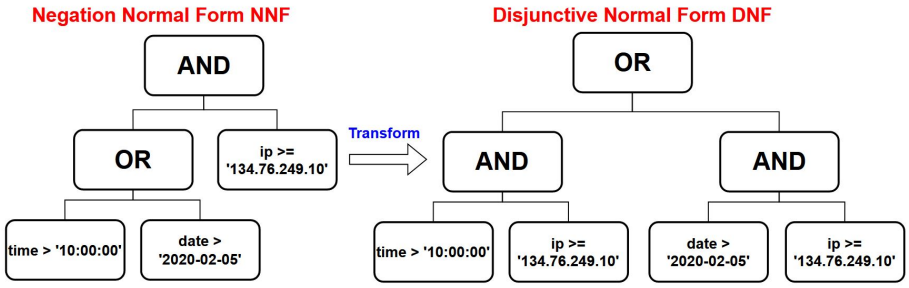


Figure 3.2 – Transforming any predicate tree of SQL like query into DNF

Since the tree is presented in DNF, the description Q_{DNF} stores the pushed-down CNF as a list of attribute restrictions (i.e., atomic predicate AP). Since, some of AP can belong to a same attribute, we propose to store and represent them with a new element, named *Range-Object*. By this way, we can group and manage efficiently AP regarding to different attributes. Such approach is motivated by object-oriented in which the operations or functions could be called or invoked by another (larger) object. Moreover, it also gives the capability to execute multi-dimensions query without exacerbating the problem of complexity in query rewriting. As a result, we present as follows a new definition in terms of specialized objects that manage query Q and semantic segment S:

Definition 3.3.1. *In the database D with its attribute set $A = \cup A_{R_i}$, a query segment $< Q_R, Q_A, Q_{DNF}, Q_D >$ or semantic segment $< S_R, S_A, S_{DNF}, S_D >$ is considered and managed as the following objects:*

1. *A Range-Object Q_{RO_j} = list of AP that belong to a same attribute $j \in A_{R_i}$.*
2. *A CNF-Object Q_{CNF} = list of Q_{RO}.*

3. A DNF-Object Q_{DNF} = list of Q_{CNF}.

To illustrate, reusing the DNF from presented query $Q2$, we assign the corresponding values for $Q2 =< Q_R, Q_A, Q_{DNF}, Q_D >$.

— $Q_R = log$.

— $Q_A = \{ip, date, time\}$.

— $Q_{DNF} = \{Q_{CNF_1}, Q_{CNF_2}\}$ where

  — $Q_{CNF_1} = \{Q_{RO_{time}}, Q_{RO_{ip}}\}$ with $Q_{RO_{time}} >' 10:00:00'$, $Q_{RO_{ip}} =' 134.76.249.10'$.

  — $Q_{CNF_2} = \{Q_{RO_{date}}, Q_{RO_{ip}}\}$ with $Q_{RO_{date}} >' 2020{-}02{-}05'$, $Q_{RO_{ip}} =' 134.76.249.10'$.

— $Q_D = null$ since query has not received the answer yet.

Each Range-Object RO_j could be expressed in different forms, such as *LowerBound, UpperBound, constant value, is null, or is not null*. It can be seen that the parentheses "(", and ")" can be replaced with the square brackets "[", and "]" respectively to support the operators "less-than-or-equal-to" ($\leq$), "greater-than-or-equal-to" ($\geq$) in atomic predicates AP. Meanwhile, the AND operators is maintained implicitly between the Range-Objects Q_{RO} in CNF-Object Q_{CNF}. We do similarly for OR operators in list of CNF-Objects Q_{CNF} in DNF-Object Q_{DNF}.

Consequently, such a kind of presented hierarchical objects within DNF converting procedure, there is no more than one Range-Object, which belongs to an attribute, that appears in a CNF-Object. For example, Q_{CNF_1} has only one instance of $Q_{RO_{time}}$. If there are two or more Range-Object that represent for an attribute in CNF, they should be merged into one. If they are unmergeable, they have to be split into two new different CNF-Objects as can be found in tree of DNF converting procedure.

3.3.3 Query Trimming

This stage is first responsible to check the satisfiability or implication relationship between a constructed query segment Q and the list of cached semantic segments S. Then, based on this relationship, Query Trimming extracts the semantic information from query segment Q to generate the corresponding probe queries PQ and remainder queries RQ. Consequently, these two presented procedures in Query Trimming are grouped into two sub-stages, Semantic Matching and Semantic Extracting, respectively.

a) Semantic Matching

We consider the Semantic Matching problem from two aspects that can be processed in parallel: *Attribute Matching* and *Predicate Matching*. More precisely, we look at how to find the relevance between attributes of query segment Q_A and semantic segment S_A. At the mean time, we examine similarly for DNF of query segment Q_{DNF} and semantic segment S_{DNF}.

In order to reach this goal, we first present the following definitions (from *Definition 3.3.2* to *3.3.5*)that correspond to Attribute Matching perspective.

Definition 3.3.2. *Given the attribute set Q_A and S_A, Common Attribute C_A is a set of attributes that are common among Q_A and S_A.*

Definition 3.3.3. *Given the attribute set Q_A and S_A, Difference Attribute D_A is a set of attributes that exist in Q but not in S.*

Definition 3.3.4. *Given the attribute set Q_A and S_A with $Q_R = S_R$, it is said that $Q_A \cap S_A \neq \varnothing$ if and only if $C_A \neq \varnothing$. Note that $\cap$ represents the intersection between Q_A and S_A.*

Definition 3.3.5. *Given the attribute set Q_A and S_A with $Q_R = S_R$, it is said that $Q_A \subseteq S_A$ if and only if $C_A \neq \varnothing$ and $D_A = \varnothing$. Note that $a \subseteq b$ means that a is contained in b.*

Example: we reuse the DNF of $Q2 =< log, \{ip, date, time\}, \{RO_{ip}, RO_{date}\}, Q_D >$ in *Definition 3.3.1*. Assuming that MASCARA contains two formatted segment $S1$ and $S2$ (as can be seen in Table 3.1). Note that they are simplified by hiding the details of their DNF. In particular, RO_{ip} and RO_{date} are not required to show in this example. The *Attribute Matching* results are shown in forms of C_A, D_A, $Q_A \cap S_A$ and $Q_A \subseteq S_A$. More precisely, $Q2_A$ has intersection with $S1_A$ meanwhile it is contained in in $S2_A$.

Index	**Simplified structure** $< S_R, S_A, S_{DNF}, S_D >$	**Relevance with Q2**			
		C_A	D_A	$Q_A \cap S_A$	$Q_A \subseteq S_A$
S1	$< log, \{ip, date\}, \{RO_{ip}, RO_{date}\}, S_D >$	$\{ip, date\}$	$\{time\}$	YES	NO
S2	$< log, \{ip, date, time, code\}, \{RO_{ip}, RO_{date}\}, S_D >$	$\{ip, date, time\}$	$\{code\}$	YES	YES

Table 3.1 – Attribute Matching between segments and query $Q2$.

In order to check the relevance between DNF of query and segments from Predicate Matching perspective, we have the following definitions (from *Definition 3.3.6* to *3.3.8*).

Recall that the predicate satisfiability between query and segment are expressed in three scenarios: *implication*, *satisfiability* and *unsatisfiability* [78]. However, presenting their matching computations is a challenge and can turn into NP-complete problem in the integer domain [43, 42, 79]. More precisely, to check *predicate implication* between Q and P with respects to *Boolean algebra* could be complex to represent due to a large of number of predicates of query and segment. Therefore, in this dissertation, we mainly focus on providing the simplified matching computations between two objects Q_{DNF} and S_{DNF}. In particular, we use $ints$ as Intersection and $diff$ as Difference for the computations.

Definition 3.3.6. *Given Q_{DNF} and S_{DNF} with $Q_R = S_R$, implication $Q_{DNF} \rightarrow S_{DNF}$ holds if and only if the entire content (i.e., all objects as Range-Objects and CNF-Objects) of Q_{DNF} is contained in S_{DNF}. The computation can be expressed by: $(Q_{DNF}\ ints\ S_{DNF})\ diff\ Q_{DNF} = null$.*

Definition 3.3.7. *Given Q_{DNF} and S_{DNF} with $Q_R = S_R$, predicate satisfiability holds if and only if a part (i.e., some objects as Range-Objects and CNF-Objects) of Q_{DNF} is answerable from S_{DNF}. The computation can be expressed by: $Q_{DNF}\ ints\ S_{DNF} \neq null$.*

Definition 3.3.8. *Given Q_{DNF} and S_{DNF} with $Q_R = S_R$, predicate unsatisfiability holds if and only if there is no overlap part (i.e., none of Range-Objects and CNF-Objects) between Q_{DNF} and S_{DNF}. The computation can be expressed by: $Q_{DNF}\ ints\ S_{DNF} = null$.*

Example: to simplify the presentation of matching between query and segment, we use a new query $Q3$ which has DNF form: $Q3 =< log, Q_A, \{RO_{ip}, RO_{date}\}, null >$. Note that Q_A and S_A are not required to be addressed in this example. In detail, Q_{DNF} consists of $RO_{ip} = (\text{'192.168.1.10'}, \text{'192.168.1.50'})$, $RO_{date} =' 2020-05-03'$. Meanwhile, MASCARA contains three segments S1, S2 and S3 in forms of $S =< log, S_A, \{RO_{ip}, RO_{date}\}, S_D >$ (as can be seen in Table 3.2). Thus, we get the results in term of Predicate Matching. More precisely, implication is held by $(Q3, S_1)$, satisfiability is held by $(Q3, S_2)$ and unsatisfiability is held by (Q, S_3).

In order to determine the relationship between Q_{DNF} and S_{DNF}, computing the Intersection $(Q_{DNF}\ ints\ S_{DNF})$ and Difference $(Q_{DNF}\ diff\ S_{DNF})$ can be seen as basic actions. Meanwhile, Implication actually consists of Intersection and Difference in its computing formula, in particular, $(Q_{DNF}\ ints\ S_{DNF})\ diff\ Q_{DNF}$. Thus, we focus mainly on the process of Intersection and Difference in this dissertation. Since our approach is object-oriented, they should be addressed from Range-Object RO to CNF-Object CNF. For example, assuming that we have the following SQL like queries:

Index	Simplified structure $< S_R, S_A, S_{DNF}, S_D >$	Relevance with Q3 $< log, Q_A, \{RO_{ip}, RO_{date}\}, null >$	
		$Q_{DNF} \rightarrow S_{DNF}$	Q_{DNF} *ints* S_{DNF}
S1	$< log, S_A$ $RO_{ip} = ('192.168.1.1','192.168.1.100')$ $RO_{date} = ('2020-05-01','2020-05-05'), S_D >$	YES	YES
S2	$< log, S_A$ $RO_{ip} = ('192.168.1.30','192.168.1.100')$ $RO_{date} = ('2020-05-01','2020-05-05'), S_D >$	NO	YES
S3	$< log, S_A$ $RO_{ip} = ('192.168.1.200','192.168.1.250')$ $RO_{date} = ('2020-05-10','2020-05-15'), S_D >$	NO	NO

Table 3.2 – Predicate Matching between segments and query $Q3$.

```
S: SELECT ip, date, time
   FROM log
   WHERE ip = '10.0.0.20'
         AND date = '2020-03-05'
         OR ip = '10.0.0.200' AND time = '12:00:00'
Q: SELECT ip, date, time
   FROM log
   WHERE ip >= '10.0.0.15' AND ip <= '10.0.0.35'
         AND date = '2020-02-05' AND time >= '10:00:00'
         OR ip = '10.0.0.100' AND time <= '08:00:00'
```

Then we can construct the objects for Q_{DNF} and S_{DNF} as shown in following Figure 3.3. The "matching" action of ROs between Q_{DNF} and S_{DNF} represents for *Intersection*, *Difference* and *Implication*. Such kind of procedure can be implemented by a nested loop over the list of CNF-Objects and Range-Objects. For example, $CNF1$ of Q_{DNF} has to match first with $CNF1$ of S_{DNF}. Then, it continues to match with $CNF2$ of S_{DNF}. It is worth recalling that the matching action operates only with RO that have the same attribute. In particular, we cannot make the intersection or difference between a RO_{ip} and RO_{date}. By this way, we mitigate the unnecessary matching over RO.

Matching action of Q_{DNF} and S_{DNF} can be expressed in different computations. Thus, we present Intersection and Difference as following since they can be seen as the fundamentals of other computations.

Intersection. The implementation of Intersection can be done efficiently by making the comparisons between either *bounds* or *exact values* of RO. Recall that, this computation has to consider also the characteristic of "(", ")" and "[", "]". The performance of Intersection depends on the number of objects in DNF since they can slow down the iteration of matching tasks. Thus, parallelization of these tasks can be seen as a solution. In

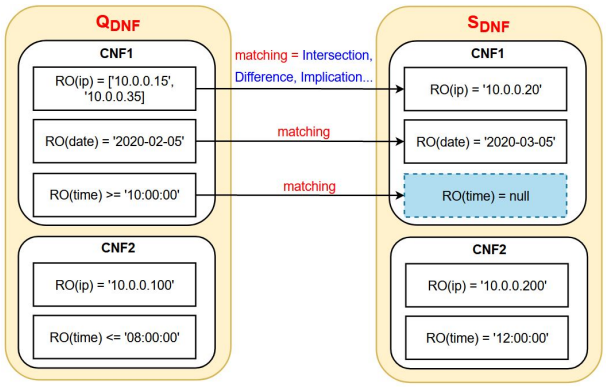


Figure 3.3 – Matching between the DNF-Object of query Q and segment S.

summary, we simplify the procedure of finding Intersection between two CNF of Q_{DNF} and S_{DNF} through Algorithm 1.

Applying this algorithm, Q_{DNF} *ints* S_{DNF} can be shown as in Figure 3.4. First we find the intersection between $CNF1$ of Q_{DNF} and S_{DNF}. Since there is no RO_{time} in $CNF1$ of S_{DNF}, we create it with *null* value. Then, intersection between $RO_{ip} =' 10.0.0.20'$ and $RO_{ip} = ['10.0.0.15',' 10.0.0.35']$ is $RO_{ip} =' 10.0.0.20'$ which is stored in $CNF1$ of the result. In opposite, we found that $RO_{date}('2020-02-05')$ *ints* $RO_{date}('2020-03-05') = null$. In other words, computing Intersection between these $CNF1$ stops due to breaking condition of loop in Algorithm 1. As a result, we conclude that $CNF1$ of Q_{DNF} and S_{DNF} have no intersection. Similarly, the computing can be done for the remaining iterations (e.g., $CNF1$ of Q_{DNF} and $CNF2$ of S_{DNF}). Unfortunately, all of the intersection results are null. Therefore, we can say that there is no intersection between Q_{DNF} and S_{DNF} which can be described formally by Q_{DNF} *ints* $S_{DNF} = null$.

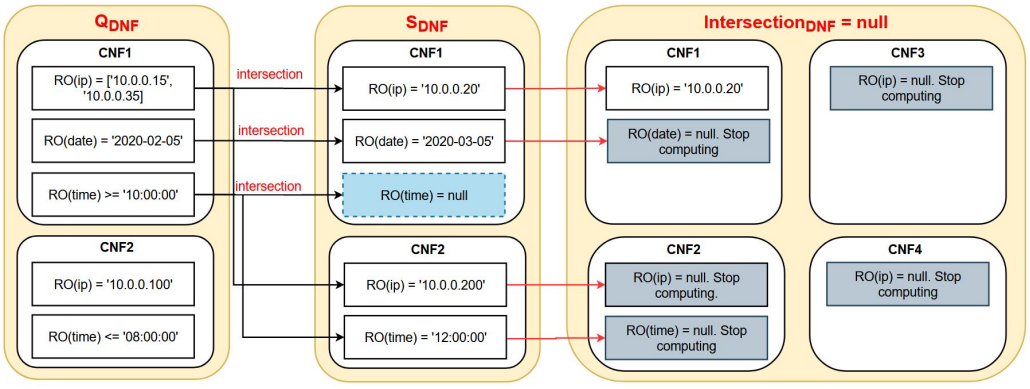


Figure 3.4 – Intersection between the DNF-Object of query Q and segment S.

Algorithm 1: Intersection between CNF-Objects

Input: $cnf\_q := Q_{CNF1}$, $cnf\_s := S_{CNF1}$
Output: cnf_rs

```
/* One of two CNF-Objects is empty (list of Range-Objects = 0)        */
if (cnf_q.isEmpty || cnf_s.isEmpty) then
    return cnf_rs
end
ro_rs := new Range-Object to store result of intersection
set_att = union set of attributes from cnf_q and cnf_s
/* Process for two Range-Objects that have the same attribute         */
for i ← set_att do
    /* Range Object RO_i of cnf_s is null                             */
    if (cnf_q.RO_i ≠ null, cnf_s.RO_i == null) then
        ro_rs := cnf_q.ROi
    end
    /* Range Object RO_i of cnf_q is null                             */
    else if (cnf_q.RO_i == null, cnf_s.RO_i ≠ null) then
        ro_rs := cnf_s.ROi
    end
    /* Two Range-Objects are not null                                 */
    else if (cnf_q.RO_i ≠ null, cnf_s.RO_i ≠ null) then
        ro_rs := cnf_q.RO_i.intersection(cnf_s.RO_i)
        /* Intersection is not null                                   */
        if (ro_rs != null) then
            /* Add intersection of range into CNF-Object result       */
            cnf_rs.addRange(ro_rs)
        end
        /* Intersection is null                                       */
        else if (ro_rs == null) then
            break the loop
        end
    end
end
```

Difference. Unlike the Intersection, computing Difference is more complicated due to the generation of additional unwanted CNFs for the result. In particular, Difference ($RO1\ diff\ RO2$) where $RO1$ with a form [LowerBound, UpperBound] and $RO2$ with an exact value can result to a decomposition in $RO1$ if value of $RO2$ is contained in $RO1$. Thus, we can have two new Range Objects $RO_{new_1} = [LowerBound, ExactValue)$ and $RO_{new_2} = (ExactValue, UpperBound]$ that are stored in two new $CNF - Object$ instead of only one as in Intersection. In the worst case, if all RO from Q_{CNF} and S_{CNF} has the

Difference, then the number of additional CNFs can grow combinatorially. The analysis for complexity of such kind of computing is also presented in works [79, 43, 42]. Therefore, we consider that Difference is the most excessive computation in Query Trimming. Note that matching tasks in such procedure can be parallelized similarly with Intersection to improve the performance. We present simplified Difference algorithm for CNF-Objects in Algorithm 2.

Algorithm 2: Difference between CNF-Objects

Input: $cnf\_q := Q_{CNF1}$, $cnf\_s := S_{CNF1}$
Output: dnf_rs

$list\_ro\_rs$:= new list of Range Object to store result of difference
/* Iteration over list of RO in S_{CNF1} */
for $i \leftarrow cnf\_s.attribute()$ **do**
  /* Get or create RO_i in cnf_q with respect to RO_i in cnf_s */
  $cnf\_q.RO_i$:= get or create new
  /* If Range Object RO_i of cnf_q is not null */
  if $(cnf\_q.RQ_i \neq null)$ **then**
    /* Computing list of Difference of two Range-Objects */
    $list\_ro\_rs := cnf\_q.RO_i.difference(cnf\_s.RQ_i)$
  end
  /* If Range Object RO_i of cnf_q is null */
  else if $(cnf\_q.RO_i == null)$ **then**
    /* Calling NOT on RO_i of cnf_q */
    $list\_ro\_rs := cnf\_q.RO_i.not()$
  end
  new_cnf := create new from $cnf\_q - RO_i$
  for $j \leftarrow list\_ro\_rs$ **do**
    $temp\_cnf := new\_cnf$
    $temp\_cnf.addRange(list\_ro\_rs.RO_i)$
    $dnf\_rs.addCNF(temp\_cnf)$
  end
end

Applying this algorithm, computing $Q_{DNF}\ diff\ S_{DNF}$ can be shown as in Figure 3.5. After finding the Difference between $RO_{ip} =' 10.0.0.20'$ and $RO_{ip} = ['10.0.0.15', '10.0.0.35']$, the results (red box) are stored in two different $CNF1$ and $CNF2$. The result of Difference (purple box) between $RO_{date} =' 2020-02-05'$ and $RO_{date} =' 2020-03-05'$ is stored in only one $CNF3$. Repeating the algorithm until reaching the end of both list CNF, we will have 12 CNFs in total to be managed in the result. Consequently, this number will grow dramatically when a query Q matches with

multiple segments S in cache.

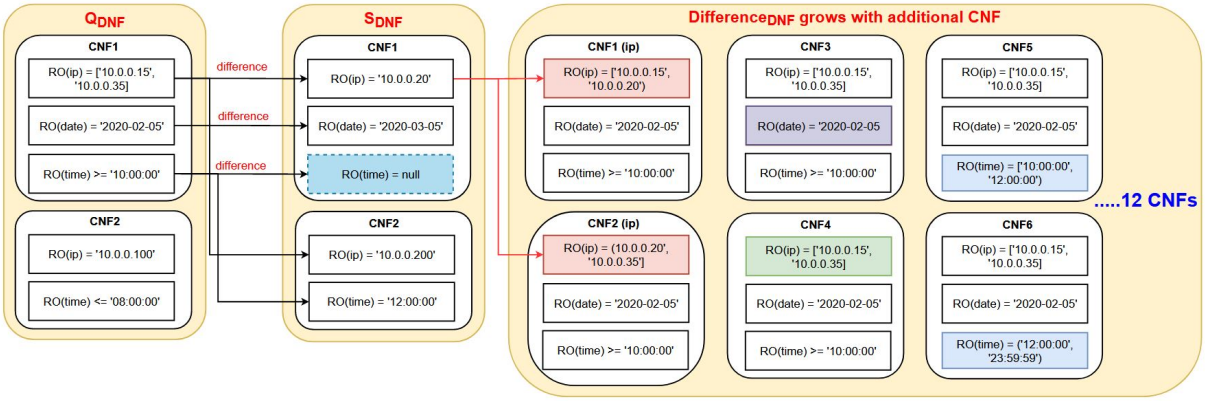


Figure 3.5 – Difference between the DNF-Object of query Q and segment S.

Based on the results from Attribute Matching and Predicate Matching, we now can verify the equivalence of Q and S. Thus, we have the following definitions for relationship of Q and S.

Definition 3.3.9. *Given a query segment Q and a semantic segment S, query implication $Q \rightarrow S$ holds if and only if two conditions are satisfied:*

— $Q_A \subseteq S_A$.

— $Q_{DNF} \rightarrow S_{DNF}$.

Definition 3.3.10. *Given a query segment Q and semantic segment S, query satisfiability holds if and only if two conditions are satisfied:*

— $Q_A \cap S_A \neq \varnothing$.

— *Sastifiability is held between Q_{DNF} and S_{DNF}.*

Definition 3.3.11. *Given a query segment Q and semantic segment S, query unsatisfiability holds if one of the two conditions is satisfy:*

— $Q_A \cap S_A = \varnothing$.

— *Unsastifiability is held between Q_{DNF} and S_{DNF}.*

Example: by adding the details of S_A for the segments and query $Q3$ from Table 3.2, we have the new Table 3.3 which shows the Query Matching in forms of Attribute and Predicate Matching. Regarding to Table 3.3, we conclude that $Q3 \rightarrow S1$, ($Q3$, $S2$) holds the satisfiability and ($Q3$, $S3$) holds unsatisfiability.

Index	Simplified structure $< S_R, S_A, S_{DNF}, S_D >$	Relevance Q3 $< log, \{ip, date\}, \{RO_{ip}, RO_{date}\}, null >$	
		Attribute Matching	Predicate Matching
S1	$< log, \{ip, date\}$ $RO_{ip} = ('192.168.1.1', '192.168.1.100')$ $RO_{date} = ('2020-05-01', '2020-05-05'), S_D >$	$Q_A \subseteq S_A$	$Q_{DNF} \rightarrow S_{DNF}$
S2	$< log, \{ip, time\}$ $RO_{ip} = ('192.168.1.30', '192.168.1.100')$ $RO_{date} = ('2020-05-01', '2020-05-05'), S_D >$	$Q_A \cap S_A \neq \varnothing$	Q_{DNF} *ints* $S_{DNF} \neq null$
S3	$< log, \{time, code\}$ $RO_{ip} = ('192.168.1.200', '192.168.1.250')$ $RO_{date} = ('2020-05-10', '2020-05-15'), S_D >$	$Q_A \cap S_A = \varnothing$	Q_{DNF} *ints* $S_{DNF} = null$

Table 3.3 – Query Matching of segments and $Q3$ from attribute and predicate perspective.

Regarding the definition of satisfiability and implication between Q and S, we present here five possible particular relations between Q_D and S_D. There exist more scenarios where S is partially or totally contained in Q, however, since these scenarios are symmetric to the case 2, 3 and 4, we can apply similar strategies. These relations are also illustrated in Figure 3.6).

1. Q_D and S_D do not match when query unsatisfiability holds with: $Q_A \cap S_A = \varnothing$ and Q_{DNF} *ints* $S_{DNF} = null$. We thus call it *Miss Matching*.
2. Q_D is contained in S_D when query implication holds with: $Q_A \subseteq S_A$ and $Q_{DNF} \rightarrow S_{DNF}$. We thus call it *Totally Matching*.
3. Q_D is vertically and partially contained in S_D when query satisfiability holds vertically with: $Q_A \cap S_A \neq \varnothing$ and $Q_{DNF} \rightarrow S_{DNF}$ We thus call it *Vertically Matching*.
4. Q_D is horizontally and partially contained in S_D when query satisfiability holds horizontally with: $Q_A \subseteq S_A$ and Q_{DNF} *ints* $S_{DNF} \neq null$. We thus call it *Horizontally Matching*.
5. Q_D is horizontally and vertically contained in S_D when query satisfiability holds partially with: $Q_A \cap S_A \neq \varnothing$ and Q_{DNF} *ints* $S_{DNF} \neq null$. We call *Mixed Matching*.

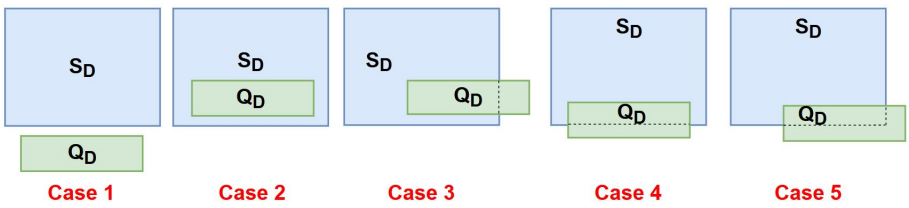


Figure 3.6 – The possible relations between Q and S that are addressed through Attribute and Predicate aspect.

b) Semantic Extracting

Receiving the results from Semantic Matching, Semantic Extracting is responsible to generate probe query PQ and remainder query RQ. Based on the presented scenarios of matching, PQ and RQ will use the results from (Q_A, S_A) and (Q_{DNF}, S_{DNF}). We present in the following the Algorithm 3 to extract and construct PQ and RQ.

Algorithm 3: Semantic Extracting

Input: $Q_< Q_R, Q_A, Q_{DNF}, Q_D >$, $S < S_R, S_A, S_{DNF}, S_D >$
Output: PQ, RQ_H, RQ_V

`/* Case: Totally matching */`
if $Q_A \subseteq S_A$ *and* $Q_{DNF} \rightarrow S_{DNF}$ **then**
  $PQ :=< Q_R, C_A, Q_{DNF}, Q_D >$
  $RQ_H := NULL$, $RQ_V := NULL$
end
`/* Case: Horizontal Matching */`
else if $Q_A \subseteq S_A$ *and* Q_{DNF} *ints* $S_{DNF} \neq \varnothing$ **then**
  $PQ :=< Q_R, C_A, Q_{DNF}, Q_D >$
  $RQ_H :=< Q_R, C_A, (Q_{DNF}$ *ints* $S_{DNF}), DMS >$
  $RQ_V := NULL$
end
`/* Case: Vertical Matching */`
else if $Q_A \cap S_A \neq \varnothing$ *and* $Q_{DNF} \rightarrow S_{DNF}$ **then**
  $PQ :=< Q_R, C_A, Q_{DNF}, Q_D >$
  $RQ_H := NULL$, $RQ_V :=< Q_R, D_A, Q_{DNF}, DMS >$
end
`/* Case: Mixed Matching */`
else if $Q_A \cap S_A \neq \varnothing$ *and* Q_{DNF} *ints* $S_{DNF} \neq null$ **then**
  $PQ :=< Q_R, C_A, Q_{DNF}, Q_D >$
  $RQ_H :=< Q_R, C_A, (Q_{DNF}$ *ints* $S_{DNF}), DMS >$
  $RQ_V :=< Q_R, D_A, Q_{DNF}, DMS >$
end
`/* Case: Miss Matching */`
else if $Q_A \cap S_A = \varnothing$ *and* Q_{DNF} *ints* $S_{DNF} = \varnothing$ **then**
  $PQ := NULL$
  $RQ :=< Q_R, D_A, Q_{DNF}, DMS >$
end

As it can be seen, PQ and RQ are constructed similarly with query Q and semantic segment S. In case of vertically and horizontally matching, we can have two types of RQ,

a vertical remainder query RQ_V and a horizontal remainder query RQ_H, respectively. These RQs are executed in DMS by compute layer with data transferred from storage layer. Interestingly, in case of Mixed Matching, we have both of them at the same time. Such kind of matching is the most complicated since the results of RQ_H and RQ_V are difficult to combined and described by only one segment. Thus, we decide keeping and processing them separately. Finally, when Miss Matching happens, one RQ, which consists of both missing attributes and predicates, is constructed . As a consequence, PQ could be executed by MASCARA as soon as they appear, meanwhile RQ has to be examined further with remaining segments in case of *multiple matching* between Q and SC.

c) Complexity of Query Trimming

While every individual segment S may contain only a small part of Q answer, multiple segments can be combined together to generate a bigger part of, or even the entire result. In other words, MASCARA has to repeatedly run Algorithm 3 over the list of the remaining segments S in SC. For example, assuming that after matching and extracting Q with S_1, we have PQ RQ_H and RQ_V. Their pointed data regions (e.g., PQ_D) are shown in Figure 3.7). Then, RQ_V will be compared with the next candidate S_2 that results in another Mixed Matching. Meanwhile, RQ_H is checked and found that it has Horizontally Matching with S_3. Continuously, Query Trimming runs to the end of the list of segments. In other words, until it cannot find any more S which can contribute to the answer of Q. As a result, in this example, we get finally three PQ (red) and three RQ (orange).

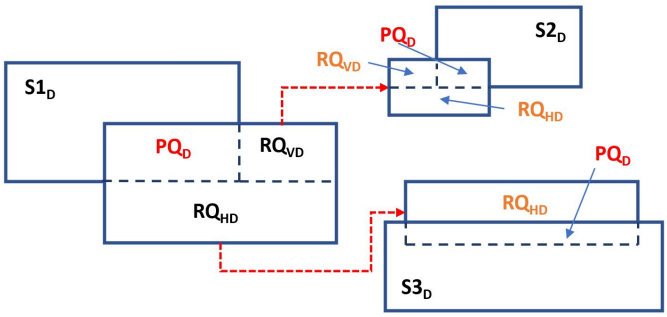


Figure 3.7 – Relationship of data regions in case of multiple matching between Q and list of S in MASCARA.

Standing for the overall complexity of Query Trimming, we have $Complexity\_QueryTrimming$. From Algorithm 3 and Figure 3.7, we consider that this stage checks implicity or satisfiability among Q and list of S in SC. Thus,

$Complexity\_QueryTrimming$ is determined by the number of segments N and the complexity of trimming algorithm $algo$ which is implemented by object oriented approach: $Complexity\_QueryTrimming = N * algo$. If single matching happens ($N = 1$), $Complexity\_QueryTrimming = algo$. It is difficult to precise the complexity of $algo$ since it depends on many different parameters, such as complexity of functions (e.g., Intersection, Difference), number of CNF, number of dimensions (i.e., attributes) in DNF, etc. In short, they can lead to an excessive computation of $algo$ which is illustrated in details through Figure 3.4 and 3.5. Specifically, with multiple matching, assuming that number of cached segment N is huge, $Complexity\_QueryTrimming$ becomes the main bottleneck in computing of MASCARA. Indeed, this issue has also been studied in [79, 43, 42]. Consequently, in this dissertation, we will revisit this bottleneck practically with respect to CPU's capability later in Section 3.4.

3.3.4 Semantic Management

a) Cache organization

MASCARA composes two parts of caching: the data region DR and the semantic segment S. Regarding with a semantic segment S, there is a pointer to the corresponding data region DR (as shown in Figure 3.8). The semantic segment can be seen as the index, meanwhile, data region is the content of the cache. Obviously, this segment structure is consistent with the formal definition of the semantic cache that we present before. In addition to the four basic components of a segment, we can add other items for maintenance, such as S_{TS} as replacement value. Since segment and data region have a strong relation, if there is an event on segment, for example, divide it to multiple segments or merge it with other segments, this also results to a relevant transformation of data region.

The conventional way to organize the query answer is to store a DR as a set of tuples (results as rows). This approach works fine for select-only queries with a key advantage of easy maintenance. For example, tuples can be added, deleted or moved between data regions conveniently. However, manipulating over the set of tuples can result in many I/O operations and large space overhead. Moreover, within the select-project queries, their results may not have the same lengths which makes the maintenances more complicated.

In MASCARA, instead of the tuple approach, we define the data structure, called block, to represent DR. In detail, every DR is stored as a linked list with one or multiple blocks where the pointer of S points to the first blocks of DR in the memory. The main

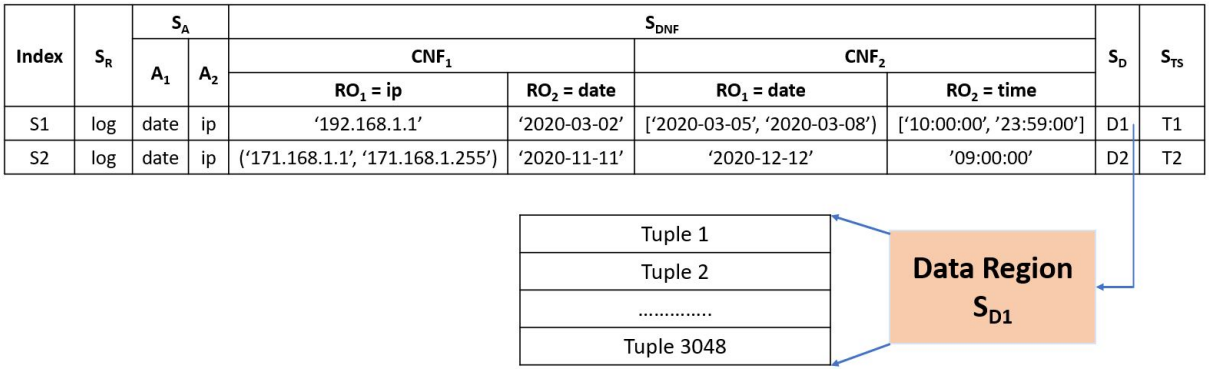


Figure 3.8 – Table of semantic description S and its corresponding data regions DR.

idea of this approach is that block has dynamic size which is manipulated through the Application Programming Interfaces (APIs) of current deployed data management systems. In other words, it entrusts the organization of data in memory to these systems, meanwhile focusing on the increased flexibility of MASCARA. For example, in Spark [8], we can build the block based on DataFrame API to facilitate the manipulation with in-memory data, such as data indexing, memory allocation, etc.

The cache space is managed at block level which makes semantic segment and its data region allocation and de-allocation more straightforward and simpler. More precisely, if there are enough free space to hold S, then we allocate the blocks to store its DR. Meanwhile, for de-allocation, the list of blocks, that represent DR, are removed from cache.

b) Replacement policy.

The main role of replacement policy is to decide which semantic segment S and its data region S_D should be removed or replaced by another one when there is insufficient space in the cache. In particular, the replacement value functions used by semantic caching can be based on temporal locality (e.g., LRU, MRU), or on semantic locality of regions (e.g., Manhattan distance). An example about using Manhattan distance is shown in Figure 3.9. With the appearance of new query $Q2$ in (b), the replacement value of $DR1$ is the negative of Manhattan distance (i.e., $rv = 2$) between the "center of gravity" of current segment ($DR1$) and the "center of gravity" of the most recent query $DR2$. Similarly, when new query $Q3$ appears in (c), replacement values of $DR1$ and $DR2$ are recalculated based on the distances from their "center of gravity" to the new "center of gravity" of $DR3$. Therefore, with this distance function, regions that are "closed" to the most recent query

have a small negative value, irrespective of when they were created, and are hence less likely to be discarded when free space is required.

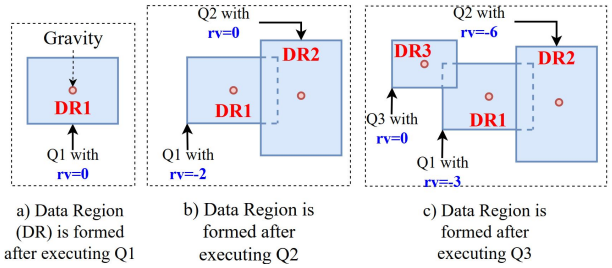


Figure 3.9 – Replacement policy with Manhattan distance.

Although the policies based on semantic locality are more efficient, they lack of generality by depending heavily on the semantic characteristics of workload. Moreover, it is mandatory to propose a method of calculating the "center of gravity". However, it results to the necessity of a cost model where many parameters should be considered, such as benefits of cached results, caching patterns, and reuse frequency of candidate. To the best of our best knowledge, SC uses Manhattan distance to enhance semantic locality is limited to few works. In particular, mobile navigation applications [30, 77] where the queries represent for the user's location and direction of motion. Consequently, to keep the generality and performance of MASCARA in different application contexts, a novel replacement value function may be proposed.

c) Coalescing strategy

To illustrate coalescing heuristic, let's examine the relationship of (Q_{DNF}, S_{DNF}) and (Q_A, S_A) in Query Trimming. We found that there is an overlap between Q_D and S_D with respect to Case 3, 4 or 5 (as shown in Figure 3.6). A question is raised here: should this part be merged into the data region of query Q_D, or should it remain in data region of segment S_D, or should it become a new independent part? Answering this question can be seen as coalescing strategies, one of the functionalities of cache management. Thus, two main approaches can be considered: Never Coalescing and Always Coalescing (as show in Figure 3.10).

Suppose that we have a sequence of three queries $Q1$, $Q2$ and $Q3$ and Always Coalescing is applied in Semantic Management. In (a), after answering $Q1$, $DR1$ pointed by segment $S1$ is created. Next, to answer $Q2$, $DR2$ can be formed, as shown in (b). Recall that data regions in cache must be mutually disjoint. In other words, $DR1$ must

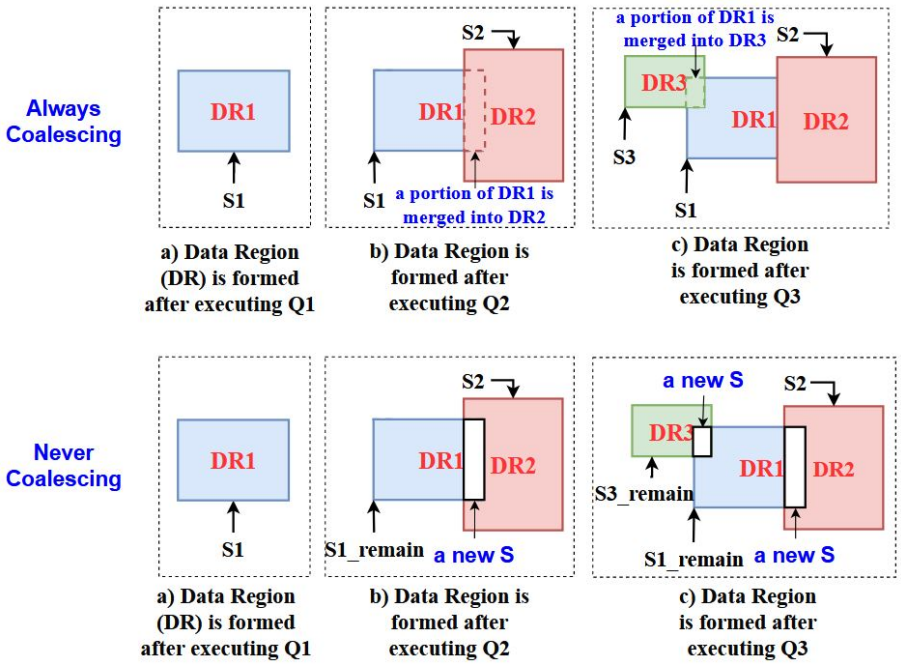


Figure 3.10 – Forming of data region in Always Coalescing and Never Coalescing.

be transformed into a new region that is disjoint with $DR2$ since they overlap. Meanwhile, $DR2$ will compose the overlapped part of $DR1$. Later, when $Q3$ is issued, a similar transformation occurs due to the overlap between $DR3$ and $DR1$ in (c). The result is a new $DR1$ that decomposes its overlapped part to merge into $DR3$. As opposite to Always Coalescing, in Never Coalescing, the overlapped part between ($DR1$, $DR2$) in (b) and ($DR1$, $DR3$) in (c) will become the new independent DR with new corresponding segments S. In addition, $DR1$, $DR2$ and $DR3$ have to transform to the new shapes without overlapped fractions.

Through this example, we consider that Never Coalescing turns the overlapped part to a new independent DR with its new corresponding S which is deduced from the intersection between Q_{DNF} and S_{DNF}. This approach reflects the frequency of reference at a finer granularity. However, the disadvantage is that it may result in a large number of small semantic segments, which increases the overhead of cache management and query processing. Meanwhile, Always Coalescing merges the overlapped part into data region of query segment Q_D. In other words, the old data region of semantic segment S_D has been split to a new one without the contribution of overlapped part. Recall that the semantic segment of this new data region is constructed from logical expression of remainder query RQ. However, when a data region DR is excessively large and it needs to be replaced

due to its low replacement value (i.e., LRU value), this results in poor cache utilization. Thus, we can say that either Always or Never Coalescing causes the problem in different aspects, such as response time, cache efficiency and cache space usage. Finally, since the approach of keeping overlapped part in data region of semantic segment S_D cannot effectively express the frequency of reference, it is not discussed in this dissertation.

3.3.5 Coalescing heuristic

By striking the balance between Always and Never Coalescing, we propose a novel coalescing strategy, named coalescing heuristic. Unlike these conventional strategies, our solution can use alternatively the mechanisms of Always and Never Coalescing by checking the current situation of data regions DR and/or their future contributions in cache. In particular, segment S which represents corresponding DR should be measured and indexed by their "profit" S_V as new element added into S. The function of calculating S_V is based on temporal (i.e., LRU) and spatial locality (i.e., contribution to query answering). Additionally, this new replacement function value can be seen as a complement of coalescing heuristic in terms of semantic management. Remember that, this can be also used in replacement policy since it consists of LRU value in its function.

Given the recency of usage, we assume that the most recent coalescing/replacement value is V_{max} , which is increased by one for each new Q. Meanwhile, the coalescing/replacement value for each data region is S_V. The process of finding the profit of a DR is divided into two steps: first, computing the intermediate profit and deciding whether to merge, based on its value, and second, computing future profit of the remaining part (as shown in Algorithm 4).

As the first step of heuristic, we measure the percentage of DR that contributes to the query answer: $p = R_Q/R$, where R_Q is the number of records that match the query answer and R is the total number of records in the region. Then, the new replacement/coalescing value is temporarily updated as follows: $S_V = S_V + (V_{max} - S_V) * p$. Note that $(V_{max} - S_V)$ ensures that the new S_V does not have a higher value than V_{max}. In other words, the new data region with respect to the last query Q will have the highest value S_V. This function is adaptable for all regions in SC, regardless whether the region contributes to answering the query or not.

Based on the updated S_V for all regions contributing to the query answer, we propose a threshold $T \in \mathbb{R}$ as a part of "coalescing filter" $S_V < Q_V * T$ that decides whether to merge all, some or none of them. In general, T can scale from 0 (*Never Coalescing*) to

Algorithm 4: Coalescing heuristic with new replacement value function

Input: Input: cache with list of S, DR and a query Q
Output: Output: cache updated with coalescing heuristic

Pass Query Trimming, outputs are: PQ, RQ
Execute PQ and RQ
`/* Replacement with the minimal ratio */`
while $r \neq$ *End Of List* **do**
  $r := S_V / size\_of\_S_D$
  Finding the minimal r
end
$Victim := S$ with DR that has minimal r
Remove $Victim$
Add RQ with new DR
`/* Heuristic of coalescing: step 1 */`
Choosing threshold T
Assign $Q_V := Vmax$
while $S \neq$ *End Of List* **do**
  $p := R_Q / R$
  $S_{V\_inter} := S_{V\_ori} + (V_{max} - S_{V\_ori}) * p$
  if $S_V < Q_V * T$ **then**
    Coalescing between S_D and Q_D
  end
  else if $S_V >= Q_V * T$ **then**
    No-coalescing between S_D and Q_D
  end
end
`/* Heuristic of coalescing: step 2 */`
while $S \neq$ *End Of List* **do**
  $pdis := S_{V\_inter} - S_{V\_ori}$
  $S_V := pdis * (1 - p) + S_{V\_ori}$
end

1 (*Always Coalescing*). Meanwhile, $Q_V = V_{max}$ is the value of the new DR with respect to the new query which appears. If $S_V < Q_V * T$, the overlapping part between existed DR and new DR of Q will be merged (coalesced) into the old one. Thus, the number of generated or cached segments are stored, resulting in an efficient response time of MASCARA. Otherwise, if $S_V >= Q_V * T$, the new DR of Q will be cached which has the same value S_V as its *predecessor DR*. By this way, this decision increases the data granularity of the cache, resulting in higher efficiency. Although we have not yet explored the cost model related to the "coalescing filter" as well as the optimization problem (e.g.,

Knapsack or Dynamic Programming [11]), we can adjust the threshold T in practical to find a "reasonable coalescing filter" based on response time, data granularity, and space utilization.

As the second step, after merging some of the DR, the remaining ones might shrink into the new parts. Therefore, their *future profit* that could be evaluated for the next queries should be recalculated as follows: $S_V = pdis * (1 - p) + S_{V\_ori}$ where S_{V_ori} is the *original profit* of the DR before starting the procedure. $pdis = S_{V\_inter} - S_{V\_ori}$, consists of the *profit distance*, is the gap between the *intermediate profit* S_{V_inter} and the *original profit* S_{V_ori}. An example of these two steps is shown in following Figure 3.11.

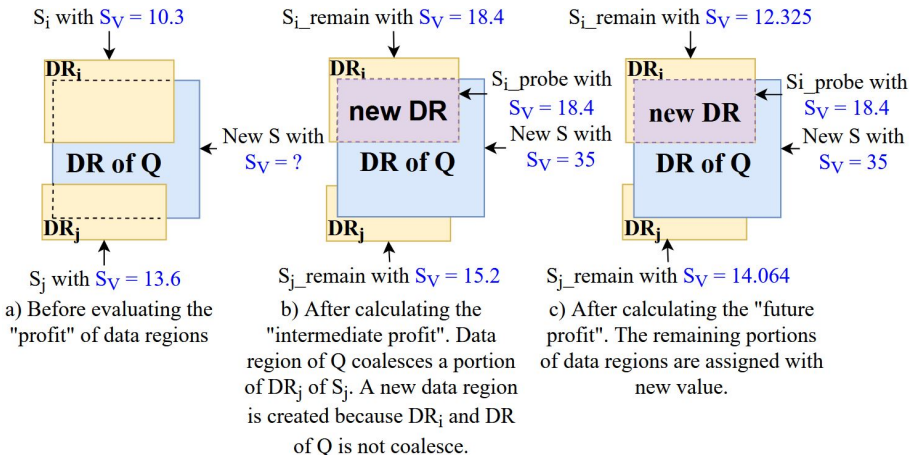


Figure 3.11 – The coalescing heuristic in cache management

We assume that the contribution of DRi and DRj to the query answer is $p_i = 0.75$ and $p_j = 0.71$, respectively. The last value $V_{max} = 35$ for the appearance of query Q. Although p_i, p_j are nearly equal, their contributions to the answer are different in size (i.e., $DRi > DRj$). From (a) to (b), the S_V of S_i has increased significantly from 10.3 to 18.4. If we choose $T = 0.5$, then $T * Q_V = 0.5 * 35 = 17.5$ does not pass the condition of "coalescing filter" ($17.5 < 18.4$). Thus, cache will keep the overlapping part between DR_i and DR_Q as a totally new one. In contrast, with the same procedure, DRj does not pass the "filter". It means that cache will merge the overlapping part of DRj into DR of Q. At the end of the procedure, in (c), we reevaluate the *future profit* of DR_i, DR_j to prepare the next Q. For example, using the formula consists of $pdis$, the *future profit* of the remaining DRi is: $S_V = (18.4 - 10.3) * (1 - 0.75) + 10.3 = 12.325$.

Since S_V is used for both coalescing and replacement, our heuristic overcomes the limitation of LRU in the context of SC. Indeed, considering that DR can vary in size,

removing it from the cache should depend not only on its contribution to the last query response, but also on its actual size. In other words, we calculate the ratio r between the actual coalescing/replacement S_V and its size s in the cache: $r = S_V/size\_of\_S_D$. Thanks to the real representation of r (e.g., $S_V = 12.325$ in the above example) if the cache needs space for a new data region DR, the selection of a victim would be more accurate than LRU. It should be noted that DR which overlaps the query response are excluded from this procedure. In summary, by approximating both *temporal* and *spatial locality*, the impact of query workload (i.e., semantic locality) could be alleviated in general applications.

To conclude, we question about the compatibility of heuristic in MASCARA. In particular, such kind of compatibility should be evaluated carefully in different metrics, such as response time, hit ratio and cache space utilization. Our coalescing heuristic may be less interesting (in response time) compared to Always Coalescing in MASCARA due to the limited throughput of Query Trimming based on CPU when complexity of query rewriting is high. However, the other benefits of coalescing heuristic, such as hit ratio and cache space usage, probably allow its contribution to be guaranteed in MASCARA. Therefore, this question will be answered practically in details through experiments in Section 3.4.

3.3.6 Multi-view processing

Up to now, Query Trimming allows to process range queries with multi dimension. More precisely, these queries consists of select and project operation in their logical expression. However, in practice, since data is structured in a large number of tables, there is a need to join the results from multiple tables based on their logical relationship. In other words, join queries should also be supported in MASCARA. In fact, query rewriting of join is more complex than select-project query due to multiple participated relations. Unfortunately, although prior SC solutions [30, 78, 52, 54, 38, 57, 80, 29, 27, 94] present in details select-project queries, none of them describes how to handle *join query* with respect to query rewriting and join result managing. A few of works from other broad line of research, *materialized view* [40, 84, 89], present new query rewriting procedure for join and aggregate query at the same time. However, their approach results to a high $Complexity\_QueryTrimming$ which reduces significantly cache performance. This result also comes from the fact that their approach stores not only the segment S but also its relationship. Moreover, since their work aims to apply SC co-located with RDMBS,

they raise a challenge of maintaining data region in physical layer. Finally, the presented algorithms are not implemented and evaluated, especially in terms of tracking and manipulating joined result. In summary, by coordinating the above elements, join query processing in MASCARA should be revisited.

To reach this goal, we first simplify the scope of study about join query in this dissertation. In particular, we focus on inner join query in which the conditions of filtering can be put in "WHERE" or "ON" clause and vice versa. Our choice is motivated by the fact that processing inner join query is not affected by the order of execution of filter and join. To explain, assuming that we have a SQL like inner join query as follows:

```
Q: SELECT users.name, users.dept, log.ip, log.date, log.action
   FROM log
   INNER JOIN users ON users.ip = log.ip
   WHERE log.ip = '109.10.10.1' AND log.date = '02-02-2022'
         AND users.dept = 'zone-7'
```

Processing this query can be done with two different approaches as illustrated through Figure 3.12. As it can be seen, the approach on left side makes join first and filters the result at the end. In contrast, the approach on right side filters first and makes join the results at the end. More precisely, it splits the original query into two sub-queries, one is executed over *user* relation and the other over *log* relation. In other words, such approach processes an inner join query as list of select-project sub-queries over different relations. Consequently, processing of these sub-queries are already presented in Query Trimming of MASCARA in Section 3.3.3.

Meanwhile, in a left join, placing a filter condition in the "ON" clause will affect a query result differently. For example, assuming that we have the two following left outer join queries:

```
Q2: SELECT users.name, users.dept, log.ip, log.date, log.action
    FROM log
    LEFT OUTER JOIN users ON log.ip = users.ip
    WHERE log.ip = '109.10.10.1' AND log.date = '02-02-2022'
          AND users.dept = 'zone-7'
Q3: SELECT users.name, users.dept, log.ip, log.date, log.action
    FROM log
    LEFT OUTER JOIN users ON log.ip = users.ip
         AND log.ip = '109.10.10.1' AND log.date = '02-02-2022'
         AND users.dept = 'zone-7'
```

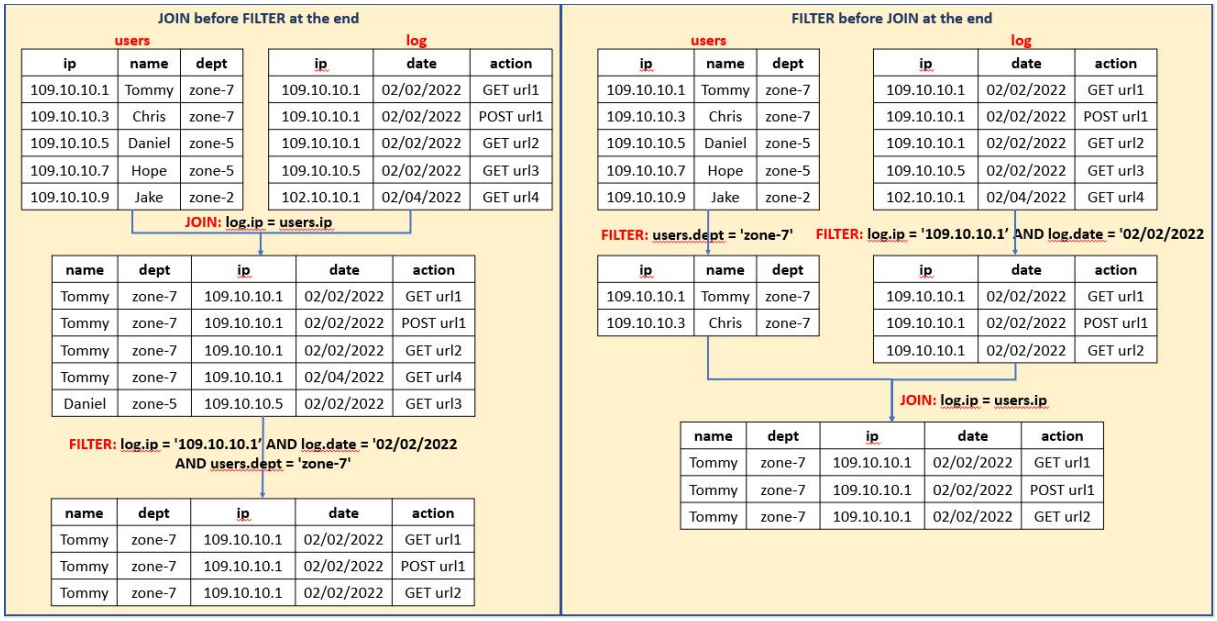


Figure 3.12 – Processing inner join query with two different approaches.

It should be apparent now that the two seemingly similar left join queries have two completely different results. In $Q2$, a filter condition was placed in the "WHERE" clause to filter records on the joined result at the end of processing. In contrast, in $Q3$, a filter condition was placed in the "ON" clause to filter the tables before starting join processing. In fact, $Q3$ can have NULL entries on the joined result whenever a match of *users* could not be made to *log*. Thus, such order of filter and join can lead to an erroneous result even though the semantic segment is well presented. Moreover, unlike inner join, left and right outer join has to process the input tables from left to right which could make query rewriting more complex. To summarize, we can conclude that such outer joins can create an issue of matching and creating corresponding semantic segment due to its seemingly similar but different results in business requirements. Therefore, to begin with, in this dissertation, we extend MASCARA with explicitly or implicitly inner join query.

Regarding the consistence result from process of inner join queries, we consider that it can be split into smaller sub-queries. Each of them consists in the filtering condition itself and works in context of corresponding relation. For example, according to presented inner join query Q, we can have two sub queries, $Q11$ on *users* and $Q12$ on *log* as following.

```
Q11: SELECT users.ip, users.name, users.dept
FROM users
WHERE users.dept = 'zone-7'
```

```
Q12: SELECT log.ip, log.date, log.action
FROM log
WHERE log.ip = '109.10.10.1' AND log.date = '02-02-2022'
```

Obviously, these two queries are Select-Project queries already supported by MASCARA. In details, they can be matched, extracted and executed through Query Trimming and Probe Query Executing respectively. Splitting query into multiple sub-queries is an action in the novel approach, called Multi-view processing, in which we propose to support inner join query.

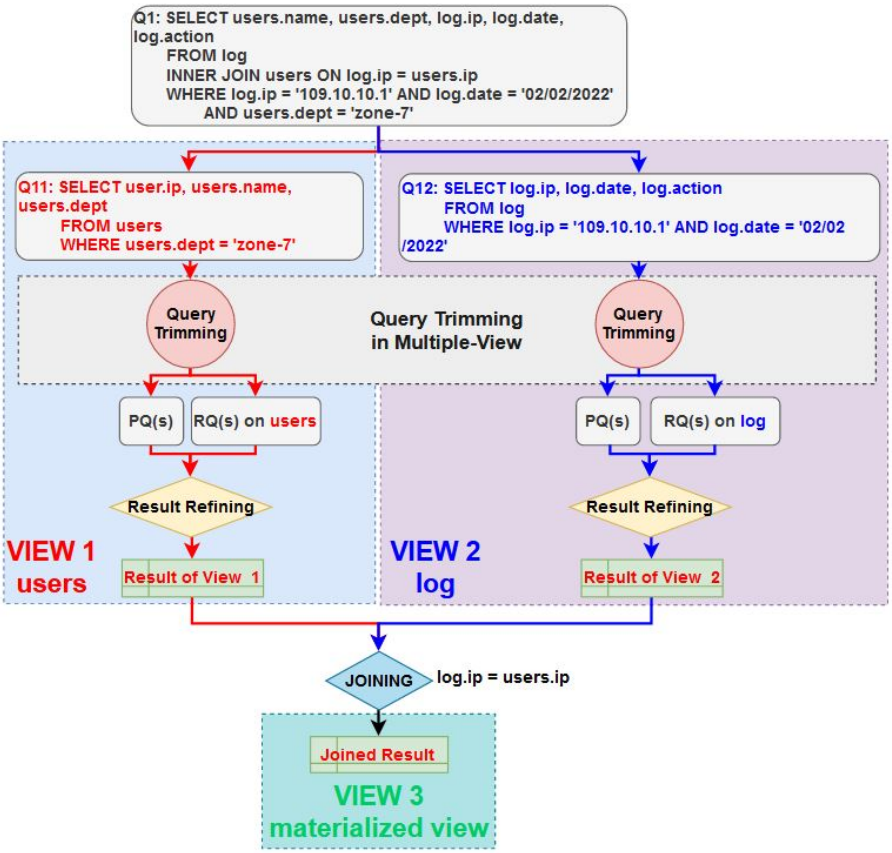


Figure 3.13 – Simplified workflow of inner join query in MASCARA with respect to Multi-view processing

We illustrate the workflow of Multi-view processing for inner query $Q1$ or $Q2$ in Figure 3.13. As it can be seen, two sub-queries run on two different views: users and log. First, Query Broking is extended to distribute attributes and predicates in original query to sub-queries. Now, each of them can be seen as a normal query which needs to match and extract

relevant semantic information from list of segments in the same view. In particular, $Q11$ needs to pass Query Trimming with list of segments that have $S_R = users$, meanwhile, $Q12$ is compared with list of segments that have $S_R = log$. Their local results are combined at the end with a join condition (e.g., $log.ip = users.ip$) to have a materialized result which is stored in the form of new data region in cache. Moreover, no matter which coalescing strategy is applied in cache management, the impact of such result to cache space seems to be similar as the other data regions from Select-Project queries.

Regarding this Multi-view, Query Trimming, which is called two times by $Q11$ and $Q12$ respectively, could greatly increase the response time due to the fact that $Complexity\_QueryTrimming$ is the bottleneck of MASCARA. In other words, applying Multi-view to process inner join query on MASCARA can cause a noticeable trade-off in response time. Obviously, this problem becomes more severe when handling multi-join conditions and multi-dimension in generated sub-queries. The compatibility of Multi-view processing with MASCARA is evaluated practically later in Section 3.4. Such evaluation allows us to prepare a plan of accelerating MASCARA on FPGA.

To sort and merge the results from sub-queries, MASCARA requires to implement a procedure, such as sort-merge-join. In general, sort-merge-join can be seen as the most effective without considering the cost of necessary pre-processing (i.e., sorting the inputs). Basically, sort-merge-join can be implemented with divide-and-conquer procedure from software perspective. However, the main drawback is that time complexity is $O(nlogn)$ since the task of sorting and merging are not executed in parallel. Thus, an input with large size can lead the sort-merge-join on MASCARA is overhead rapidly.

Since the sub-queries can run independently in Query Trimming, it is essential to multiply table of semantic description by number of views in query. For example, if we have segments from two sub-queries $Q11$ and $Q12$, we need to organize two semantic tables: first table stores segments on users and the second table is for log. Recall that these tables have structure described in details by Figure 3.8 in Section 3.3.4. To maintain the relationship of segments $Q11$ and $Q12$ in two tables, we provide a new pointer S_M in the segment (as shown in Figure 3.14).

As it can be seen, $Q11$ of table *users* link to $Q12$ of table *log* through a pointer. Thus, the data region for materialized result is represented by multiple segments. Furthermore, both $Q11$ and $Q12$ keep the connection to their data region. By this way, if a new query has one view (relation), it requires to check only either table users or log. Interestingly, it is possible to take a corresponding portion of this data region without unnecessary running

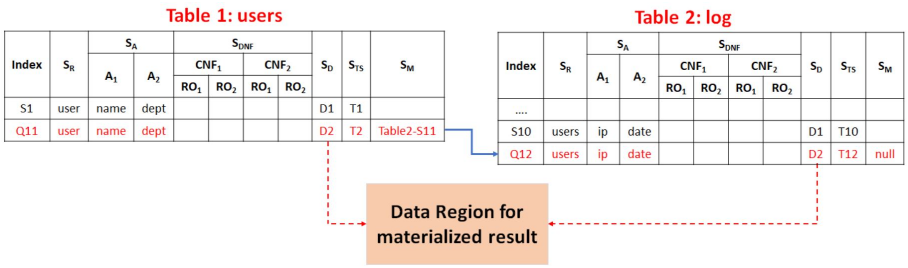


Figure 3.14 – Managing multiple semantic description tables within Multi-view processing.

Query Trimming on both tables. Last, to ensure the consistency between segments of table, we consider that the timestamps of linked segments are synchronized for cache replacement.

3.3.7 Result Refining

In order to merge the results from PQ and RQ, it is necessary to have a data structure to express the relationship between every part of query segment Q and the involved semantic segments S. In other words, combination of results seems to be the puzzle issue in which we need to identify the position, the order and the type of combining. Therefore, to reach this goal, in Result Refining stage, we use a binary tree, named *Query Plan Tree (QPT)* which is referenced in [78].

Each non-leaf node of QPT represents a type of combination, such as vertical combining VC and horizontal combining HC with respect to the type of matching we have. Meanwhile, a leaf node can be the result of PQ on a specified segment S, or RQ_V or RQ_H on server. The QPT is constructed from top-to-bottom in parallel with the Query Trimming stage. More precisely, a node as well as its ancestor or children will be created if and only if its relative Query Trimming was finished. Finally, to combine the results, we need to traverse the QPT. Using the presented example in Figure 3.7, a QPT can be constructed simultaneously with Query Trimming stage as shown in Figure 3.15.

Consider the case when Q has Mixed Matching with $S1$. After passing the semantic extraction, the root of tree, node $N1$, is created with label HC, which means that the results of the sub-trees must be appended horizontally. Also, $N1$'s sub-trees represent the two trimmed parts: left sub-tree for Horizontal Matching between Q and $S3$, and right-sub tree for Mixed Matching between Q and $S2$. Thus, node $N2$ with HC and $N3$ with VC are created for these two sub-trees. Until now, the left sub-tree of $N2$ cannot be further trimmed, it stands for the PQ_D that is executed over data region $S3_D$. Its RQ_H will be

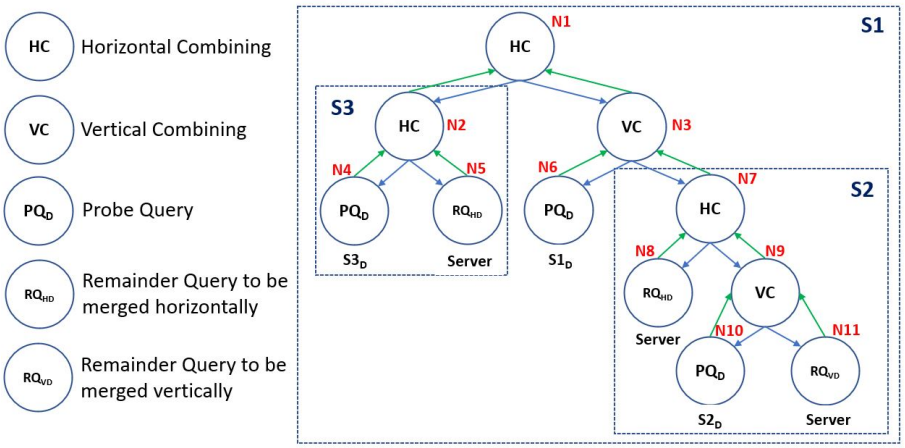


Figure 3.15 – A query plan tree with respect to mutiple matching in Query Trimming.

processed by the DMS to return the result RQ_{HD}. In opposite, the QPT grows up with its right-sub tree $N3$ in which we get the remaining nodes, such as $N8$ (RQ_{HD}), $N7$ (PQ_D), etc. After having all the results of nodes, we can combine them to get the final answer by traversing the QPT with respect to the green arrows.

The time to generate this QPT depends on the time complexity of Query Trimming $Complexity\_QueryTrimming$ for N segments. Meanwhile, time to traverse it to combine the result depends on the time to process PQ and RQ, and the time to execute the combine action VC or HC. In our study, time of query execution which depends on relational evaluation and combine action can be simplified no matter the size of results. Consequently, through examining the QPT of Result Refining, we can say that the improvement of SC performance can be done mainly by accelerating Query Trimming and PQ execution. Meanwhile, acceleration of RQ as black-box between the compute and the storage layer of DMS, is not coped in this dissertation.

3.4 Validation

In this section, we present the experimental results with respect to the following objectives. First, we want to compare the performances of MASCARA with the other solutions, such as No-Cache and Block-Cache. Second, we also show the limitations (e.g., dimension of DNF, multi-view processing) that can reduce the performance of MASCARA. Thirdly, we show the unsuitability of our coalescing heuristic for MASCARA based on CPU where we need to prioritize accelerating the performance (i.e., response time). Based on these

above elements, we raise the question of hardware acceleration for MASCARA in different aspects, such as query rewriting for select-project-join query, coalescing strategies and probe query executing. To conduct the experiments, we present the benchmark environment, data sets as well as the impact factors in Section 3.4.1. Then, we give the results of our solutions in Section 3.4.2.

3.4.1 Validation environment

a) Configuration

We conducted all experiments on a server consisting a single node which is equipped with one Intel® Xeon® Gold 5118 CPU running at 2.30GHz. Moreover, this server has 64GB of RAM. The software stack of this system consists of Linux Ubuntu 16.04.4 Long Term Support, Hadoop 2.6.0 [85], and Spark 3.0.0 [8], and MASCARA.

The compute layer of DMS on server is deployed with a query engine, Apache Spark 3.0.0 [8], to execute the remainder queries. Spark can be seen as a general-purpose cluster computing engine with libraries for streaming, graph processing, machine learning and SQL. Moreover, SparkSQL for database processing, provides DataFrame API that can perform relational operations on different formats in various kinds of storage. In order to run the remaining queries, Spark is configured to run in a "Standalone Mode", a simple cluster manager incorporated with Spark. We use the default configuration for the amount of memory and CPU cores in Spark to execute the remaining queries. In details, it will take up to 12 cores of the server node and 64GB RAM to distribute to corresponding executors. Here, we configure only one Java Virtual Machine (JVM) Executor with all 12 cores.

Meanwhile, for the storage layer of DMS, we use Hadoop Distributed File System (HDFS) [85] to cooperate with SparkSQL, that provides reliable, scalable and fault tolerant data storage on specialized hardware. In addition, HDFS is designed to hold large amount of data and provides faster access to data from Spark SQL. We use the default chunk or block size of HDFS, in particular, 128MB for each block that is replicated three times.

Since MASCARA is the middleware layer of DMS, we implement it (in Java) on top of Spark to execute the probe queries. This prototype can be named as MASCARA-Server. Query Trimming of MASCARA is implemented to be processed with (logical) multi-threads. Each thread is invoked to run the comparison between a pair of available CNF

between Q and S. However, internal tasks or functions, in particular, range comparison and read/write of CNF are not executed in parallel since implementing them on CPU can be seen as a challenge in terms of task and event management. Similarly, generated PQ from Query Trimming can be executed in parallel by multi-threads. Meanwhile, sort-merge in sort-merge-join can run with three threads that correspond to three joined relations *lineitem*, *orders*, and *customer*. To summarize, in our experiment, we fix to have 24 (logical) threads running in parallel over 12 cores of CPU.

It is reasonable enough to study the performance and main bottlenecks of MASCARA (i.e., Query Trimming) in single node. The reason is within a server node, data communication between the compute (i.e., Spark) and the storage layer (i.e., Hadoop) is still expensive. This issue is even worse in case of we extend to a cluster of computing nodes where the restriction of network bandwidth emphasizes more the role of MASCARA.

b) Datasets and workloads

We use the data set that is generated by TPC-H benchmark tool kit[1]. Using the integrated tool DBGen, we can generate different sizes of tables by configuring the Scaling Factor (SF). For example, $SF = 1GB$ means that we generate a data set with many tables which total size is 1GB. In our experiments, we need only three main relations, such as *lineitem*, *order* and *customer* that consumes approximately 84% of total size. Moreover, the tuples of this *lineitem* table is computed by $bs\_tuple * SF$ where $bs\_tuples = 6,000,000$. Similarly, *bs_tuples* of *order* and *customer* are equal to $1,500,000$ and $150,000$. As consequence, this data set will be loaded in HDFS with the default configuration blocks (i.e., 128MB).

To create the workload, we customize two queries of TPC-H, Q6 and Q5 [1]. Our choice is motivated by the fact that Q6 heavily utilizes the filtering operation and also is I/O intensive (with storage layer). In other words, executing for a large number of Q6(s) can clarify the role of caches. Meanwhile, Q5 can be seen as a typical join query which runs over the three largest relations of TPC-H: *lineitem*, *order* and *customer*. Additionally, their logical expression are also modified by adding more attributes (as multi-dimensions) and CNFs (in terms of atomic predicates). Such customization facilitates the analysis of MASCARA's performance and bottleneck. In short, Q5 is suitable to analysis the capability of MASCARA with respect to Multi-view processing.

Regarding to the schema of the three relations, we assume that the candidate keys are always requested in queries. Furthermore, since MASCARA supports multi-dimension

queries, it allows to request multiple attributes that are either integer or decimal in relations. For example, with three dimensions (3D) query, in *lineitem*, we can have *l_linenumber*, *l_quantity* and *l_shipdate* in the condition of customized Q6. Thus, a customized Q6 can be generated as following:

```
SELECT *
FROM lineitem
WHERE l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
      AND l_quantity <= 24 and l_quantity > 10
      OR l_linenumber = 120 AND l_shipdate > '1997-02-02'
```

Specifically, in customized Q5, we make an explicit or implicit inner join query between three presented tables. We also consider that such kind of query always consists of joined key in the projection. Thus, a customized Q5 can be generated as follows:

```
SELECT *
FROM customer, orders, lineitem
WHERE c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND o_orderdate >= '1994-01-01'
      OR l_shipdate >= '1994-01-11' AND l_quantity <= 24
```

Each run consists of a sufficient number of queries to warm up firstly the cache, based on the requirement of cache initialization. Then, for each measurement, a workload of 500 queries will be taken. Consequently, we obtain the final result to present each experiment by an average of five times running.

c) Metrics

We measure the main metrics, such as total response time, hit ratio with respect to semantic locality of workload, and data transferred from the storage layer in order to exhibit the benefits of MASCARA. Moreover, we also consider cache space utilization to compare different coalescing strategies in semantic management.

d) Impact factors

The performances of MASCARA could be affected by many factors. Regarding to the presented metrics, the impact factors as shown in Table 3.4.

Impact factors	Description	Affected procedure	Affected metric
CNF_Seg	Number of CNF in segment	Query Trimming RQ Execution PQ Execution	Response Time
Dim_Seg	Dimension of segment Or number of attributes in DNF		
CNF_Query	Number of CNF in query		
Dim_Query	Dimension of query		
Nb_Seg	Number of segments		
SF	Scaling factor of data set		
SK	Skew of query accesses to Hot Region	Cache Management PQ Execution RQ Execution	Hit Ratio Response Time
HR	Hot Region of data set		
Size_C	Size of cache		

Table 3.4 – Impact factors of MASCARA's performance

As it can be seen, we have a group of impact factors that are related to the complexity of queries or segments, such as CNF_Seg or Dim_Query. Obviously, this group affects the execution time not only in Query Trimming but also in PQ and RQ. For example, increasing the Dim_Seg or CNF_Seg in segments can result to an excessive computation in Query Trimming. Moreover, if we have RQ at the end of Query Trimming, its form could be complex and can thus results in a high latency in execution.

Meanwhile, the second group affects hit ratio and transferred data from server. With respect to the size of cache $Size\_C$, the hit ratio can be improved in case cache is large enough to hold always frequently accessed data. Thus, according to the increase of hit rate, cache can save the data required from the server. Besides the size of cache $Size\_C$, we have two other factors: SK for Skew and HR for Hot Region. More precisely, HR represents the data that is frequently requested by the queries. In complement with Hot Region, Skew is a fixed fraction of the queries in the workload that access around the center point within the Hot Region. For example, Figure 3.16 shows that the access pattern is skewed 80% with Hot Region for 40% of the data set. Thus, it can be seen that 80% of the queries in workload will access this Hot Region. Moreover, scaling SK to generate queries over predefined Hot Region HR of the data set can be seen as such a kind of simulation for semantic relevance between queries.

In fact, there are many other impact factors that are not listed in the Table 3.4. In particular, regarding to the Query Trimming's output, we can have a list of probe queries PQ to be executed over their associated data regions DR in cache. The execution of this list can consume much more time than expected due to its size (i.e., number of generated

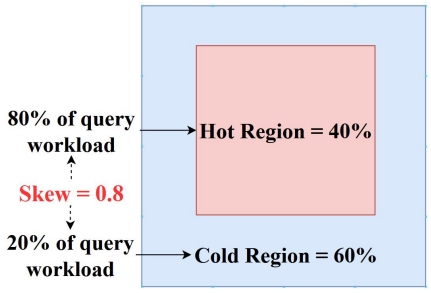


Figure 3.16 – An example about Hot Region of data set and Skew in query generator.

PQ). PQ could be processed in parallel depending on the resources of the server node. Moreover, assuming that the data region DR are large, thus, scanning over them in PQ without using an index technique could increase the latency of the system.

Regarding to inner join query processing, we propose to add a new factor, called Nb_Table, which represents the number of participating relations in the query. Moreover, the size of tables from views to be joined, called $Size\_View_RS$, impact the execution of sort-merge-join of MASCARA.

e) Evaluated prototype

We use three prototypes to evaluate the performance: No-Cache, Block-Cache and MASCARA-Server (MAS-Server). Since the comparisons of Semantic with Page and Tuple caching were done in [30, 78], we propose to use another simplified prototype to analysis side-by-side with MAS-Server. In details, we present Block-Cache which stores only the data without any semantic information of queries. Unlike MAS-Server, it needs to compare all data in cached blocks to the condition of query rather than checking the equivalence of query's logical expression. Thus, it supports only two types of matching: Totally Matching and Miss Matching. Obviously, if a query matches with multiple blocks, then its results, to be cached, could be duplicated with other existing blocks. Block-Cache can be constructed based on the caching mechanism of the compute layer (i.e., Spark) and added policies and strategies corresponding to the block's characteristic.

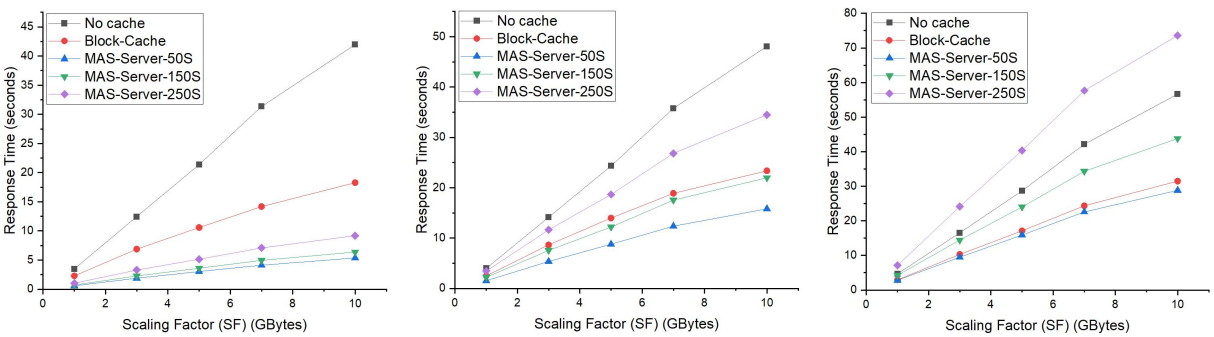
Furthermore, we can extend MAS-Server with different coalescing strategies, such as Always Coalescing (AC), Never Coalescing (NC) and Coalescing Heuristic (CH). In complement, the conventional strategies work with Least Recently Used (LRU) policy, except the heuristic which uses its own "profit" value.

3.4.2 Experimental results

In this section, we present first the overall performance (in response time) of MASCARA. Then, we study in details the partition of execution time for each computing modules. It is worth noting that Multi-view processing is also evaluated together. By this way, we can identify the suitable components to be accelerated by hardware later. Other benefits of MASCARA are also presented through hit ratio and data transferred. Finally, we explain why our coalescing heuristic is not compatible to be used on MASCARA based on CPU through reevaluating the performance with different strategies.

a) Performances

We present response time of running workloads of Q6 customized in Figure 3.17 in which we change *Dim_Query*, *SF* and *Nb_Seg* since they can affect significantly the performance of MASCARA.



(a) $Dim\_Query \leq 3$ (b) $3 < Dim\_Query \leq 5$ (c) $5 < Dim\_Query \leq 8$

Figure 3.17 – Response time of Q6 mod

For each evaluation, the workloads are generated by skewing up to 0.9 (i.e., Skew $SK = 0.9$) within a Hot Region $HR = 10\%$ of the relation *lineitem*. By this way, the cache size is large enough to entirely store HR, thus, increasing hit ratio for both Block-Cache and MAS-Server.

We need to warm up the cache through a sufficient number of warm-up queries that depends on the running context. In particular, MAS-Server-50S means that MAS-Server should be initialized with 50 segments before running the experiments. We do similarly for two other testing scenarios, MAS-Server-150S and MAS-Server-250S. Meanwhile, the Block-Cache is warmed-up by using the same workload of MAS-Server-50S. Finally,

MAS-Server applies Never Coalescing strategy as baseline of semantic management. We present in details the value of factors for queries and segments, such as CNF_Query or CNF_Seg in Table 3.5.

Impact factors	Value
CNF_Seg	vary from 40 to 50 CNFs in semantic segment
Dim_Seg	vary with dim ≤ 3, $3 <$ dim ≤ 5, $5 <$ dim ≤ 8
CNF_Query	vary from 40 to 50 CNFs in query segment
Dim_Query	vary with dim ≤ 3, $3 <$ dim ≤ 5, $5 <$ dim ≤ 8
Nb_Seg	50, 150, and 250 segments initialized in cache
SF	vary from 1GB to 10GB for dataset
SK	0.9 of Hot Region
HR	10% of lineitem
Size_C	30% of lineitem

Table 3.5 – Details of (Select-Project) queries and segments for evaluating response time of MASCARA-Server

Evaluation of Q6. In Figure 3.17a, with $Dim\_Query \leq 3$, the response time of No-Cache is the highest because all of the queries are executed by the server where cost of data transfer from the storage to the compute layer is expensive. Moreover, this response time increases dramatically when scaling factor $SF = 10GB$. Meanwhile, the Block-Cache has a better response time although its hit ratio is less than 40% (more details in Section c)). In particular, Block-Cache is 1.5 and 2.3 times faster than No-Cache when $SF = 1GB$ and $SF = 10GB$, respectively. Block-Cache does not have Partial Matching, thus, it has low hit ratio and its benefit is limited in terms of executing many RQ in DMS. Moreover, the cached answer in Block-Cache can be duplicated since missing record(s) in block requires a new block from storage Thus, this problem can degrade cache space usage.

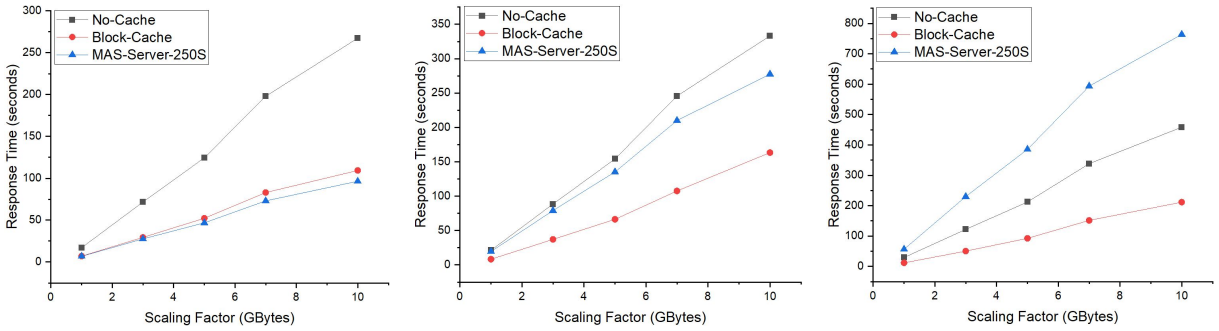
Our MAS-Server runs faster than Block-Cache with all scenarios. This acceleration comes from its high hit ratio (up to 90% approximately, more details in Section c)) when cache size $Size\_C$ is large enough (e.g., 30% of data set). By this way, MAS-Server generates less RQ that are expensive to be executed. We consider that MAS-Server-50S is the fastest one because its Query Trimming seems to be less expensive (with $Nb\_Seg = 50$) compared to the others. For example, with $SF = 1GB$, MAS-Server-50S is 6.0 and 3.9 times faster than No-Cache and Block-Cache, respectively. In opposite, response time of MAS-Server-250S is increased due to its high number of initialized segment, $Nb\_Seg = 250$. In particular, it can result to be only 4.6 and 2.0 times faster than No-Cache and Block-Cache respectively with $SF = 10GB$.

Regarding to Figure 3.17b, the performances of MAS-Server are reduced. In detail, the speed-ups of MAS-Server-50S compared to No-Cache and Block-Cache, are only 2.6 and 1.6 times respectively with $SF = 1GB$. This phenomenon is even worse for MAS-Server-250S. More precisely, it is only 1.2 times faster than No-Cache with $SF = 1GB$. Interestingly, we found that MAS-Server-250S is slower than Block-Cache. Although MAS-Server-50S and MAS-Server-150S still remain better than Block-Cache, their performances reduce dramatically. The reason is the complexity of Query Trimming, in particular, Dim_Query and Dim_Seg that could be up to five attributes. Indeed, MASCARA allows to handle Query Trimming by multiple threads of comparison CNF between Q and S. However, the internal task of CNF, in particular, Range-Object comparison (with if-else) statement can be executed in pipelining fashion. Therefore, when Dim_Query and Dim_Seg increase, multiple threading of CPU is not capable to handle, unless concurrency of threads is considered.

As shown in Figure 3.17c, in all of three scenarios, MAS-Server runs slower than Block Cache when greater number of dimension (i.e., $5 < Dim \leq 8$). Notably, the response time of MAS-Server-250S is obviously slower than No-Cache due to the fact that a bottleneck can appear in Query Trimming. It is worth to note that the growth of dimension in query does not only degrade Query Trimming but also leads to the high latency of PQ and RQ due to their generated complex format. Thus, response time of MAS-Server-250S can be even worse. Consequently, the worst case of MASCARA is when it has a large number of complex segments in Dim and DNF that turns to be the main bottleneck of computing as shown in MAS-Server-250S.

Evaluation of Q5. To continue, we present the performance of MASCARA when running workloads of customized Q5(s) (as shown in Figure 3.18). We focus only on MAS-Server with 250 segments, named MAS-Server-250S, where Multi-view processing creates the overhead of computing (i.e., Query Trimming multiple times). It is worth to note that Dim represents the dimension of *lineitem*. Meanwhile, the other dimensions from *orders* and *customer* are fixed with three and one, respectively since they have a limitation of integer or decimal attributes.

We found that MAS-Server can improve the execution performance compared to No-Cache and Block-Cache if the number of dimensions is small (i.e.,$Dim\_Query \leq 3$) (as shown in Figure 3.18a). Otherwise, its response time increases dramatically (as shown in Figure 3.18b and 3.18c). For example, with $SF = 10GB$ and a high number of dimensions (i.e., $5 < Dim \leq 8$), it runs 1.67 and 3.61 times slower than No-Cache and Block-Cache,



(a) $Dim\_Query \leq 3$ (b) $3 < Dim\_Query \leq 5$ (c) $5 < Dim\_Query \leq 8$

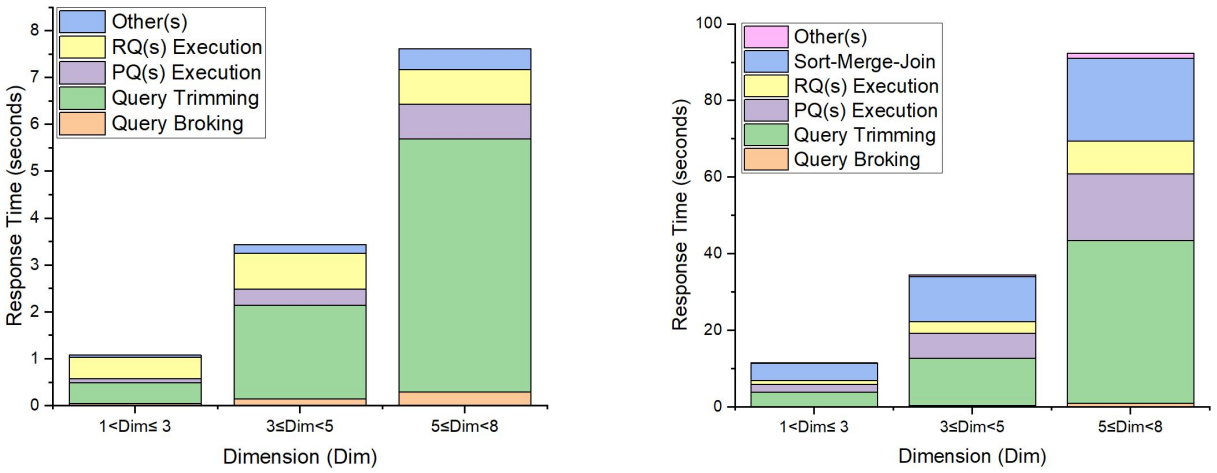
Figure 3.18 – Response time of Q5 mod

respectively. To explain, MAS-Server (i.e., MAS-Server-250S) deploys Multi-View processing with multi-threads on CPU. Thus, each thread is responsible to run Query Trimming for a relation. When a thread runs, it invokes the sub-threads to handle computation of Query Trimming. However, they are not efficient in processing overlapped tasks as we explained before in overall evaluation of Q6. As a result, Query Trimming for a relation can meet the overhead quickly, especially when the number of dimensions of Q and S increase (i.e., $5 < Dim \leq 8$). Moreover, sort-merge-join of MASCARA at the end of Multi-view processing is not optimized since it relies on a divide-and-conquer algorithm which traverses the viewed results sequentially. Consequently, Multi-view concept enables query rewriting for join query in MASCARA, however, it is not compatible with MASCARA-Server because of lack of task parallelism and an efficient sort-merge-join operator.

b) Partitions of response times

In order to find out the potential of acceleration, it is better to know in details about the execution of each stage or procedure in MAS-Server. Thus, regarding to Figure 3.17, we illustrate the partition of execution time for workload of Q6 and Q5 respectively within MAS-Server-250S and $SF = 1GB$. Note that the total execution time of the partitions in Figure 3.19 can be greater than the end-to-end execution time in Figure 3.17 because some of the processes overlap or work in parallel. For example, the execution on a list of PQ has been launched as soon as they appear while waiting the return from remainder queries RQ running from DMS.

In Figure 3.19a, Query Trimming is always the highest partition compared to other procedures. In particular, it consumes 41.9% of the time when $1 \leq Dim < 3$ and in-



(a) Partition of workload Q6(s).

(b) Partition of workload Q5(s).

Figure 3.19 – Partitions of execution time on MAS-Server-250S.

creases up to 70.9% of the time when $5 \leq Dim \leq 8$. In other words, this is the result of increasing query complexity through dimension of query and segment. We also notice that the execution time of PQ consumes in average 8.9% of the time since a large number of them can be generated with respect to the multiple matching between Q and the list of S. Generally, execution time of PQ can also increase due to its complex logical expression with respect to dimension of Q and S. Meanwhile, we notice that RQ takes a large time partition (i.e., 41.87%) when Q and S are not complex (i.e., Dim is less than 3). This partition is reduced significantly (i.e., to 9.6% of the time) when Query Trimming has more Dim in computing (i.e., Dim is greater than 5). In other words, when Query Trimming is complex enough, it overcomes the RQ execution to be the main bottleneck of MASCARA-Server. However, the partition of RQ execution over total response time can regain when SF is high (e.g., $SF = 100GB$). It is even worse where the storage layer is a cloud service where communication with the compute layer is expensive. This issue can be alleviated if hit ratio of MASCARA is high to minimize the operations from DMS.

Other elements, in particular, Query Broking and a group of small executions, such as replacing segments in the cache, combining results, etc., take small partitions since their complexities can be negligible as explained before in this chapter. Finally, Result Refining, which runs in parallel with Query Trimming, is not necessary to presented here.

In Figure 3.19b, we see Query Trimming still occupies the largest partition, about $32.5 - 45.9\%$ in total response time of MASCARA-Server. Within Multi-view processing, the number of generated PQ increases due to the fact that processing of three relations.

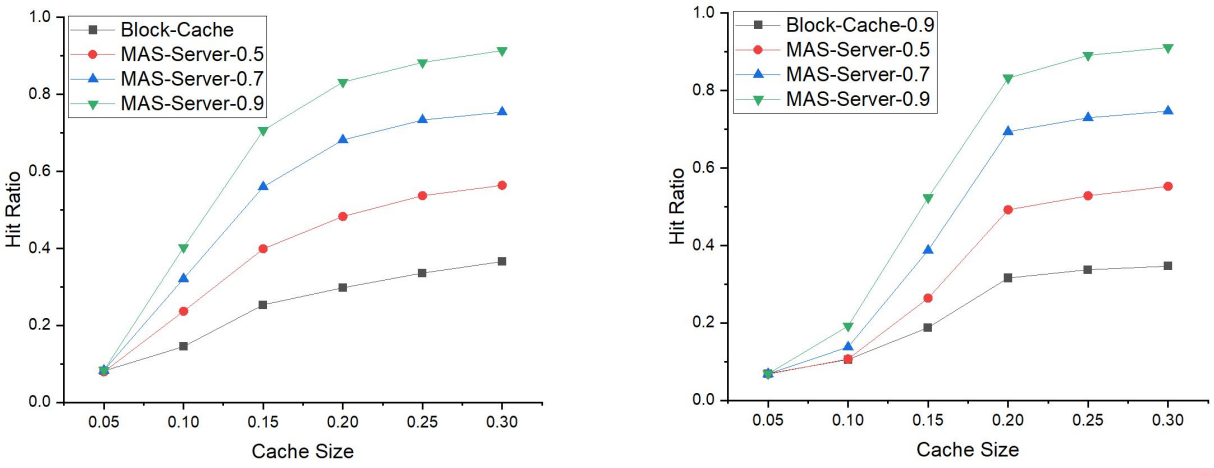
Thus, the partition of PQ execution consumes more than it used to be for Q6(s) (in average 18.8% of total time). Interestingly, with the emergence of sort-merge-join in MASCARA-Server, RQ execution now takes a small part (i.e., 8.6% of total time) for $SF = 1GB$. Indeed, sort-merge-join with non-optimal divide-and-conquer algorithm, can consume up to 39.1% of total time when viewed results are less than 5% of the size of the data set. It is worth noting that the execution time of such operation can increase significantly in case sizes of generated viewed results are larger.

To conclude, excluding RQ execution, we consider that Query Trimming, PQ execution and sort-merge-join are the elements that consume large proportions in MAS-Server based on CPU. Fortunately, they can be accelerated with an appropriate specialized hardware (i.e., FPGA) thanks to the modular approach of MASCARA. To do it, we expect to offload the computations of Query Trimming that are represented in modules of MASCARA into the hardware accelerators. Moreover, developing necessary database operators, such as filter, project and sort-merge-join would be relevant to maintain the execution of PQ and allow Multi-view processing. In contrast, RQ is executed by DMS and depends heavily on its formats, query engine and storage system. It then turns into the issue of query optimization that could be studied by different approaches, such as query plan optimization, management of data partitions, or even acceleration by FPGA. In this dissertation, the processing on DMS is seen a black-box and thus is out of the scope of our dissertation.

c) Hit ratios

Hit ratio the metric which can be used to enhance the performance of MASCARA by alleviating the emergence of RQ execution in DMS. Hence, it is meaningful to compare the hit ratio of MAS-Server with Block-Cache. To do that, we need to examine different impact factors. First, the semantic locality of the workload which is represented by Skew SK of generated queries and Hot Region HR of relation. Second, size of cache $Size\_C$ which varies from small to large (as shown in Table 3.6) can limit the maximum hit ratio that can be achieved. As a result, we illustrate the hit ratio of MAS-Server with Never Coalescing and other solutions in Figure 3.20.

As it can be seen, we run MAS-Server with three scenarios (i.e., workload of Q6s), $SK = 0.5$, $SK = 0.7$ and $SK = 0.9$. In opposite, with the Block-Cache, we apply the best scenario (i.e., $SK = 0.9$) to compare with our MAS-Server. Meanwhile, size of cache is initialized based on size of data set, in particular, table *lineitem*. Other factors of



(a) With $HR = 5\%$

(b) With $HR = 10\%$

Figure 3.20 – Hit Ratio

workload and segments in MAS-Server can be found in Table 3.6.

Impact factors	**Value**
CNF_Seg	from 40 to 50 CNFs in semantic segment
Dim_Seg	$3 < \text{dim} \leq 5$
CNF_Query	from 40 to 50 CNFs in query segment
Dim_Query	$3 < \text{dim} \leq 5$
Nb_Seg	150 segments are initialized in cache
SF	1GB for data set
SK	vary with 0.5, 0.7 and 0.9
HR	10% of lineitem
Size_C	vary from 5% to 30% of lineitem

Table 3.6 – Details of queries and segments for evaluating hit ratio of cache

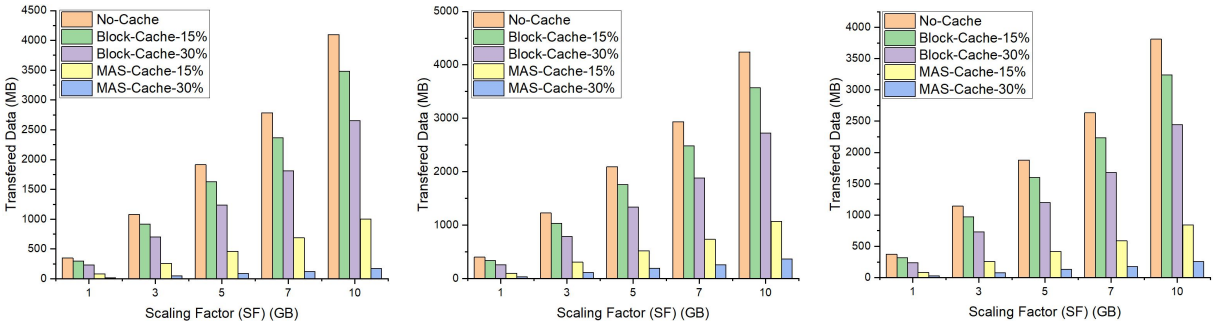
In Figure 3.20a, where the cache size $Size\_C$ is small, hit ratios of all scenarios are very low. In particular, with $Size\_C = 5\%$, hit ratios are 8.2%, 7.9%, 8.3% and 8.3% for Block-Cache, MAS-Server-0.5, MAS-Server-0.7 and MAS-Server-0.9, respectively. The reason is that cache size is too small to keep the data of Hot Region consistently, thus, replacement happens more frequently. When cache sizes are large enough, the hit ratios increase. For example, $Size\_C = 15\%$, hit ratios are 25.3%, 39.9%, 55.9% and 70.6% for Block-Cache, MAS-Server-0.5, MAS-Server-0.7 and MAS-Server-0.9, respectively. Obviously, hit ratio of Block-Cache is small, even if $SK = 0.9$, since it does not support partial matching. Thus, MAS-Server can have higher possibility to find out entire or partial of answers when

SK increases. In particular, with $SK = 0.9$, hit ratio of MAS-Server-0.9 can reach 90%. Other scenarios have less possibility to have overlap queries, such as MAS-Server-0.5 and MAS-Server-0.7, they thus have lower hit ratios, 56.4% and 75.3%, respectively.

In Figure 3.20b, when $HR = 10\%$, the cache size needs to be equal to 10% or larger than 15% of relation (i.e., *lineitem*) to have a good hit ratio. More precisely, from $Cache\_S = 20\%$, hit ratios of MAS-Server increase dramatically. For example, they are 49.2%, 69.4%, and 83.2% for MAS-Server-0.5, MAS-Server-0.7 and MAS-Server-0.9 respectively. Consequently, hit ratio of MAS-Server, even if with low semantic relevance (i.e., $SK = 0.5$), outperforms the Block-Cache when cache size is large enough to store Hot Region.

d) Data transferred

Expensive data communication between the compute and the storage layer in DMS can reduce dramatically the performance of query processing, especially in distribute systems or with a cloud service storage. Another metric we want to measure is the transferred data from the storage to the compute layer in case MASCARA does not find any match between Q and SC (as shown in Figure 3.21).



(a) With $3 \leq$ Dim (b) With $3 <$ Dim ≤ 5 (c) With $5 <$ Dim ≤ 8

Figure 3.21 – Data needs to be transferred.

We use workload of Q6(s) which $SK = 0.9$ and $HR = 10\%$ to measure the data transferred. Moreover, each Block-Cache and MAS-Server have two testing scenarios, with $Cache\_S = 15\%$, and $Cache\_S = 30\%$ of relation. Other parameters can be found in Table 3.7.

As it can be seen, No-Cache has the worst results while it executes queries repeatedly without reusing previous data. In other words, it does not save any data to be transferred

Impact factors	Value
CNF_Seg	from 40 to 50 CNFs in semantic segment
Dim_Seg	vary with dim ≤ 3, $3 < \text{dim} \leq 5$, $5 < \text{dim} \leq 8$
CNF_Query	from 40 to 50 CNFs in query segment
Dim_Query	vary with dim ≤ 3, $3 < \text{dim} \leq 5$, $5 < \text{dim} \leq 8$
Nb_Seg	150 segments are initialized in cache
SF	vary from 1GB to 10GB for data set
SK	0.9
HR	10% of lineitem
Size_C	vary with 15% and 30% of lineitem

Table 3.7 – Details of queries and segments for evaluating transferred data of cache

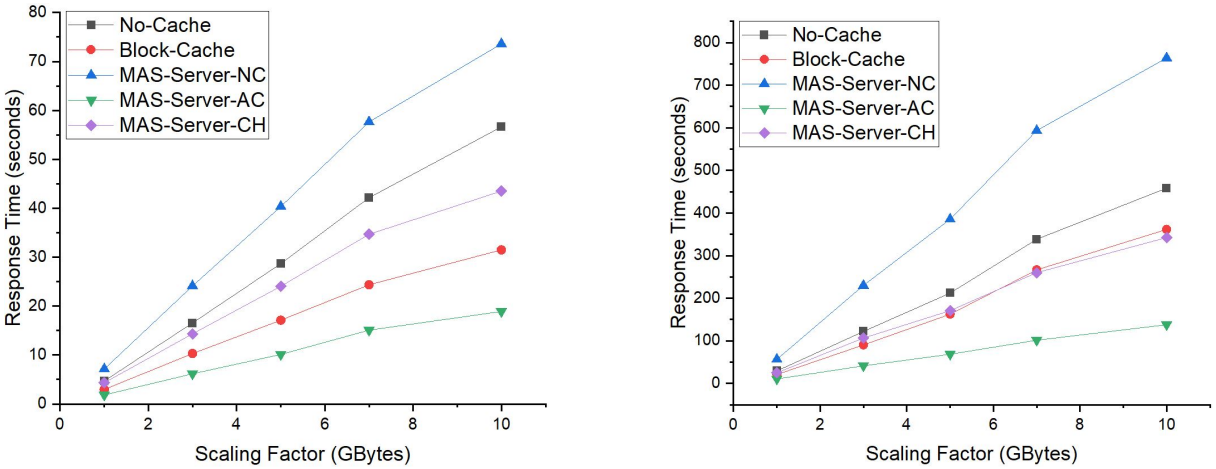
from the storage layer. Meanwhile, Block-Cache reduces the volume of transferred data thanks to its caching mechanism (i.e., Totally Matching). For example, in Figure 3.21a, with $SF = 1GB$, Block-Cache-15% reduces 14.9% of answer while Block-Cache-30% reduces 34.7% of answer compared to No-Cache. However, when a MISS happens, it requires to take full answer from storage instead of considering only the missing data. Moreover, having only Totally Matching, thus, hit ratio of Block-Cache is not high (as presented in Section c)) which limits the data can be saved.

In opposite, MAS-Server saves a significant amount of data thanks to its capability of partial answering. This thus leads to high hit ratio. In particular, with $SF = 1GB$, it can save up to 75.6% and 95.8% of answer with MAS-Server-15% and MAS-Server-30%, respectively, compared to No-Cache. These results are still the best when increasing the dimension Dim (as shown in Figure 3.21b and 3.21c). It is worth noting that, despite the configuration of SK and HR, the correlation between generated queries can vary randomly. More precisely, the number of overlapped attributes is three but the attributes may vary. It thus results in different amounts of transferred data when changing the dimension of queries of segments. In summary, MAS-Server saves the most amount of data transferred from the storage layer thanks to its highest hit ratio regarding to the semantic locality addressed by $SK = 0.9$ and $HR = 10\%$.

e) Performances with coalescing heuristic

We analyze in the following how much performance (i.e., response time) can be gained from applying different coalescing strategies in semantic management of MASCARA-Server (as shown in Figure 3.22). Our motivation is to show the unsuitability of coalescing heuristic for MASCARA-Server where we need to prioritize accelerating the performance

(i.e., response time).



(a) Workload of Q6(s). (b) Workload of Q5(s).

Figure 3.22 – Response time of different coalescing strategies

In this experiment, we use workload of Q6(s) and Q5(s) with respect to the parameters presented in Table 3.6. In particulars, we focus mainly on the issue of multi-dimensions and large number of segments since they are the bottlenecks of Query Trimming in MASCARA-Server. Thus, we initialize Dim between five and eight attributes and $Nb\_Seg = 250$. Meanwhile, we fix a balance value $T = 0.5$ for threshold when applying coalescing heuristic.

In Figure 3.22a, NC has the worst results. As already discussed in Section a), response time of NC is also slower than No-Cache and Block-Cache due to its large number of processed segments and multi-dimensions in Q and S. Meanwhile, AC exhibits the best result since it generates less number of segments. For example, with $SF = 1GB$, MAS-Server-AC is 2.5, 1.6, and 3.9 time faster than No-Cache, Block-Cache and MAS-Server-NC, respectively. Our heuristic CH gives better results than NC but it is still worse than Block-Cache. Indeed, although the number of generated segments in CH is reduced, CPU still meets the overhead of computing in Query Trimming. Additionally, MAS-Server-CH is 2.3 times slower than MAS-Server-AC. In other words, CH does not exhibit a convincing enough result compared to AC which is more preferable in MASCARA-Server. This issue is confirmed when observing the results in Figure 3.22b. In details, with $SF = 1GB$, MAS-Server AC is 5.5 and 2.4 times faster than MAS-Server-NC and MAS-Server-CH, respectively.

To conclude, we consider that response time of MAS-Server has the highest priority

compared to other metrics, such as data granularity and cache space usage. More precisely, MAS-Server requires AC to mitigate the issue of Query Trimming which is considered as the computing bottleneck. In other words, AC which brings more acceleration than NC and even our heuristic CH, is preferable to be applied on MAS-Server in big data application. However, let's assume that MASCARA will be accelerated soon by hardware accelerators, especially with Query Trimming. In other words, Query Trimming would not be the main bottleneck anymore in such kind of acceleration. Thus, the priority of AC could be reduced due to its drawback in hit ratio. Meanwhile, CH which exhibits balanced results in different aspects may become the chosen. Therefore, the response time and other benefits, such as hit ratio and cache space usage of CH, will be revisited when MASCARA will be accelerated.

3.5 Conclusion

This chapter presents a *ModulAr Semantic CAching fRAmework (MASCARA)* as a middleware layer in architecture of a new data management system (DMS). The main objective of MASCARA is to provide semantic cache-as-service by dividing and regrouping the functionalities, computations and procedures of semantic caching (SC) into modules and stages. By this way, we increase the flexibility, scalability and adaptability of MASCARA with respect to the change of requirements, environments and infrastructures. Moreover, MASCARA can be seen as our foundation of SC performance improvement partly or entirely by using I/O device accelerators. The best case shows that MASCARA is up to 3.9 times faster than the baseline (i.e., block cache). Meanwhile, it can have a hit ratio up to 91% and thus save approximately 95% data transfer from the storage layer when the cache size is large enough with respect to the semantic locality of workload.

By striking the balance between Always and Never Coalescing, we propose a novel coalescing strategy, named *coalescing heuristic*. Unlike these conventional strategies, our solution can use alternatively the mechanisms of Always and Never Coalescing by checking the current situation of data regions DR and/or their future contributions in cache. In short, this heuristic can decide to coalesce data regions based on temporal locality and spatial locality that are represented through a new replacement value function. However, the experimental results show that Coalescing Heuristic is 2.4 times slower than Always Coalescing. Thus, we consider that such heuristic is not compatible with MASCARA based on CPU as Always Coalescing. Indeed, it is urgent to alleviate the bottleneck of

Query Trimming in MASCARA by reducing number of generated segments as done in AC. Therefore, if the excessive computing of Query Trimming is solved or accelerated (i.e., by FPGA), coalescing heuristic could able to compete with Always Coalescing, especially its hit ratio and cache space usage.

We also present an approach for query rewriting of select-project-join in MASCARA, named *Multi-view processing*, which decomposes an original (inner) join query into (select-project) sub-queries that belong to different joined relations or views. In other words, instead of processing a (inner) join query, we process a list of sub-queries and join their results at the end. Enabling join processing in the cache may lead to handle many generated sub-queries and thus may be expensive. Moreover, the sort-merge-join procedure of results from views has also a negative impact on the total response time. The experimental results show that performance of MASCARA based on CPU can be reduced significantly. In particular, it runs 1.7 and 3.6 times slower than No-Cache and Block-Cache when the dimension of the query and the number of segments is high. Thus, we expect the drawback of Multi-view processing can be solved when performance of Query Trimming in MASCARA is enhanced, for example, with hardware accelerators on FPGA.

In fact, MASCARA shows a significant decline in performance with the bottleneck in computing of Query Trimming. Therefore, in the next chapter, we will propose a cooperative model between MASCARA-FPGA which accelerates not only the Query Trimming but also the other modules, such as filtering-projecting and sort-merge-join. Such model is expected to overcome the problem of Multi-view processing in MASCARA by leveraging the high throughput of FPGA accelerators. Finally, we will revisit the benefits of coalescing heuristic within MASCARA-FPGA, such as total response time, hit ratio and cache space usage. By doing so, we acknowledge that the heuristic is compatible to be used with FPGA acceleration.

Chapter 4

MASCARA-FPGA

Abstract: The previous chapter presented ModulAr Semantic CAching fRAmework (MASCARA) as middleware layer of DMS in order to handle the problem of query re-execution. Nevertheless, MASCARA is less efficient when Query Trimming grows combinatorially with the a large number of segments in cache and high number of dimensions in query. To overcome this limitation, this chapter aims to use FPGA as a solution to accelerate the performance of MASCARA. Moreover, FPGAs have shown more efficiency for almost all application domains against MPP with respect to big data . Therefore, we present MASCARA-FPGA, a cooperative model, to accelerate not only the Query Trimming but also generated probe queries regarding to MASCARA. In details, we design the relevant accelerators for Query Trimming on FPGA with respect to bottom-to-top pipeline execution. We also develop the essential database (DB) operators on FPGA to execute the generated sub-queries. Additionally, we provide the other components to link MASCARA and FPGA, such as FPGA-Adapter and Query Process Controller (QPC). Since MASCARA-FPGA can handle the drawbacks of coalescing heuristic for semantic management and multi-view processing for (inner) join query, we revisit their benefits on MASCARA-FPGA. Firstly, we show an appropriate acceleration (in response time), high hit ratio and low cache space usage of the heuristic. Secondly, we prove that MASCARA-FPGA overcomes the bottleneck of Multi-view in processing inner join queries.

4.1 Introduction

Although MASCARA can help to solve the issue of fine grained data usability in DMS, it shows a significant decline in performance due to the complexity of Query Trimming. The reason is that finding the intersections and differences between queries and segments in Query Trimming grows combinatorially with the number of segments and dimensions of query [43, 42, 79]. Moreover, maintaining and manipulating a large number of segments that have relationship, become complex and cumbersome. More specifically, regarding to big data, a cache can hold *several thousands or even millions* of segments that are presented in multi-dimensions, the aforementioned problem is expensive to solve. Thus, we doubt that the problem of Query Trimming in MASCARA could be handled well by scaling-up the hardware resources of compute tier (e.g., CPUs, RAMs) since they will soon reach "power wall" limitation [9] when processing a huge number of computing iterations.

In order to overcome this limitation, this chapter aims to use FPGA as a solution to accelerate the performance of MASCARA. Our choice is motivated by the fact that FPGA has shown more efficiency for almost all application domains against MPPin the domain of big data [36]. Moreover, with a rising demand for parallelizing algorithm, in particular, Query Trimming, it is worth looking a combination between MASCARA and FPGA. Nevertheless, to the best of our knowledge, most of the presented schemes of FPGA with respect to DMS, were proposed purely in accelerating in-memory database (DB) or providing specialized DB accelerators [81, 86, 95, 71, 91, 32, 63, 13, 88]. In other words, there is no state-of-the-art work on FPGA which considers the integration and acceleration of SC.

Therefore, we present a novel approach, named MASCARA-FPGA cooperative model, to accelerate not only the Query Trimming but also range query processing. To achieve this goal, we design the query rewriting of MASCARA and their tasks with respect to FPGA accelerators in bottom-to-top pipeline execution. We also develop the essential database (DB) operators on FPGA to execute the generated sub-queries from query rewriting, such as filter, project and sort-merge-join. We organize cache on off-chip memory (i.e., DRAM) of FPGA which supports a reasonable capacity and high bandwidth connection to accelerators. Besides the main components of MASCARA on FPGA, we also provide some modules to bridge the gap between high-level services and low-level accelerators, such as FPGA-Adapter and Query Process Controller (QPC).

We recall that MASCARA based on CPU limits the benefits of two other contribu-

tions, coalescing heuristic for semantic management and multi-view processing for (inner) join query. Currently, with the high throughput accelerators within MASCARA-FPGA, we re-evaluate that their benefits can be increase. On the one hand, we show that the acceleration of MASCARA-FPGA with CH is guaranteed meanwhile CH can bring other benefits, such as hit ratio and cache space usage. More precisely, by using "profit" as replacement value, hit ratio of the heuristic can be boosted up while keeping an appropriate number of segments. Additionally, cache space usage can be reduced by removing duplicated key attributes in data regions. On the other hand, by deploying multiple accelerators of Query Trimming to run in parallel, MASCARA-FPGA overcomes the bottleneck of Multi-view processing and exhibits a high acceleration with workload of inner join queries. In other words, MASCARA-FPGA now allows to process inner join query rapidly.

Consequently, the contributions of this chapter can be summarized as follows:

1. We present an architecture of cooperation between MASCARA and FPGA where the accelerators and their tasks can work in parallel.
2. We design Query Trimming accelerators to overcome the bottleneck of excessive computation, for example, Intersection and Difference in terms of Predicate Matching.
3. We implement basic DB operators, such as filter, project and sort-merge-join, to handle a large number of generated probe queries on FPGA.
4. We extensively analyze the response times of MASCARA-FPGA with the data sets of the TPC-H benchmark [1], especially, the speed-up which can be increased from each accelerator individually.
5. We reevaluate the benefits of coalescing heuristic in MASCARA-FPGA within different aspects, such as response time, hit ratio and cache space usage.
6. We revisit and prove that the bottleneck of Multi-view processing for (inner) join query in MASCARA is overcomed by high throughput accelerators on FPGA.

The chapter is structured as follows. In Section 4.2, we introduce the architecture and the main components of MASCARA-FPGA. We then present a specialized adapter, in Section 4.2.2, to bridge the gap between high-level interfaces of MASCARA with low-level functions on FPGA. Next, we present the accelerators of Query Trimming in Section 4.2.3. Later, in Section 4.2.5, we discuss the implementation of the DB accelerators to process select-project-join queries. We evaluate the performance of MASCARA-FPGA in

Section 4.3. At the same time, we analyze the benefits of FPGA accelerators to overcome the bottleneck of Multi-view processing and coalescing heuristic. Finally, we make the conclusion and present challenges of MASCARA-FPGA in Section 4.4.

4.2 MASCARA-FPGA

4.2.1 Principals

As can be shown in Figure 4.1, this section describes the cooperative model MASCARA-FPGA as well as its major components that contribute to the integration and operation of accelerators on FPGA. It is worth to note that MASCARA-FPGA is developed, in terms of the architecture where FPGA is used as an IO accelerator.

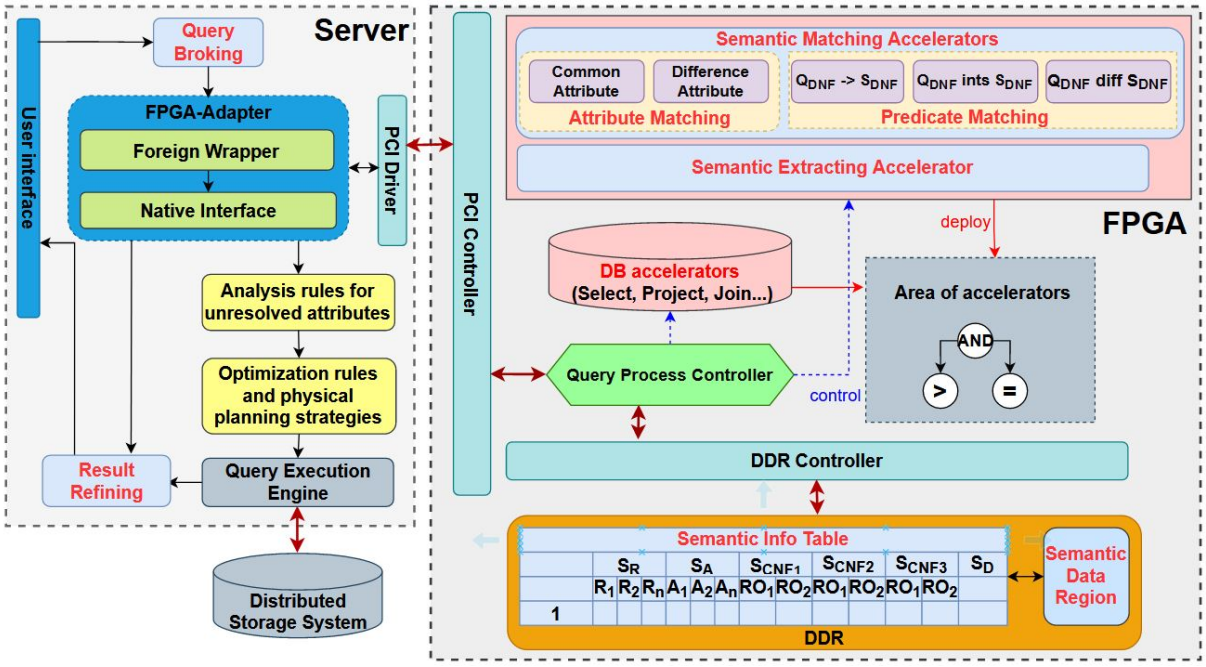


Figure 4.1 – Cooperative model MASCARA-FPGA with its major components.

Generally, the workflow of query processing in MASCARA-FPGA is similar to MASCARA-Server, except that the Query Trimming and its outputs (i.e., list of probe queries) are done on FPGA. In other words, the stages of MASCARA now are distributed in two different sides, server and FPGA. More precisely, we keep Query Broking and Result Refining in server side while Query Trimming and Semantic Management are offloaded into FPGA. Since the throughput of Query Broking does not cause any problem

to MASCARA, CPU's capability of server is reasonable enough to maintain it. Anyway, the answers of remainder queries are generated by DMS on the server side and combined together with results coming from FPGA later. It is thus meaningful to deploy Result Refining on server before returning the final answer to the user. Note that, to execute remainder queries with data coming from the storage layer, we can use a data processing framework (e.g., Spark [8]) which can consist of multiple steps for optimization and execution (as presented by yellow blocks in Figure 4.1). These steps could be expanded and customized to accelerate the execution of remainder queries through several sets of defined rules which handles different phases of query execution: analysis, logical optimization, physical planning, and code generation.

Due to the fact that MASCARA serves as a middleware layer in DMS, it should be developed with a high-level programming language, like Java or Scala that operates in Java Virtual Machine (JVM). In opposite, the accelerators can be created by High-Level Synthesis (HLS) languages (e.g., C++14) [53]. To leverage the functionalities on FPGA, the need of Java-to-native bridge is essential in our model. To overcome this problem, we use a component called FPGA-Adapter which can encapsulate or wrap the native functionalities into high-level interfaces. By this way, the communication between MASCARA and FPGA can be handled without being forced to rewrite the code.

To maintain the communication between stages of MASCARA from server to FPGA, Peripheral Component Interconnect express (PCIe) which has an extremely high bandwidth, can be used. As a part of such communication, PCI Driver and PCI Controller are mandatory to control bus specific functions and support issuing standardized transactions. Recently, the new generation of FPGA developing platform, for example, Xilinx Vitis HLS [53], can handle the communication through PCIe automatically.

In opposite to Query Broking and Result Refining, Query Trimming now is offloaded and accelerated on FPGA. In detail, the two sub-stages, Semantic Matching and Semantic Extracting are converted to corresponding accelerators. For example, Semantic Matching category can consist of the accelerators (also called kernels) for two main functions: Attribute Matching and Predicate Matching. It is worth to note that a kernel can be divided into smaller accelerated engines to increase the level of task parallelism on FPGA. In particular, the kernel Predicate Matching can be split to three different functions as engines: *Intersection* (Q_{DNF} *ints* S_{DNF}), *Implication* ($Q_{DNF} \rightarrow S_{DNF}$) and *Difference* ($Q_{DNF}$ $diff$ S_{DNF}). Meanwhile, Attribute Matching won't be split since its computation for relation between Q_A and S_A through Common Attribute C_A and Difference Attribute

D_A is straightforward.

After passing Semantic Extracting, the outputs, list of probe queries will be stored temporally in a buffer and executed in parallel depending to the availability of database (DB) operators on FPGA. Such list can consist of many primitive functions to construct the relevant DB kernels, such as scan, filter, combine, etc.

A small component called Query Process Controller (QPC) is responsible for the management of the returned signals, statuses and results within the workflow of Query Trimming and execution of PQ on FPGA. Moreover, it can have a Finite State Machine (FSM) if necessary to handle the complex PQ when MASCARA-FPGA will be extended for other inputs, such as aggregate or top-n queries. Before running, the kernels are initialized with a number of sufficient instances, called Computing Units (CUs) that are deployed physically on different dedicated regions on FPGA. To facilitate the development, HLS tools like Xilinx Vitis HLS [53] or Catapult HLS [5], can be used.

As complement of the accelerators, we also organize and manage SC on off-chip memory (i.e., Dynamic Random Access Memory DRAM) of FPGA. In fact, with a large scaled data set (i.e., 10GB), on-chip memory (i.e., Block RAM BRAM) of FPGA is not appropriate obviously to cache the data since its capability is too small. Thus, an alternative approach on DRAM is preferable thanks to its larger spaces. Moreover, it also provides a high bandwidth connection from cache to kernels through a DDR Controller.

4.2.2 FPGA-Adapter

In this dissertation, we consider that MASCARA is developed in Java/Scala while FPGA accelerators are developed in C++14 through a particular HLS platform (i.e., Vitis HLS). Thus, FPGA-Adapter should be constructed to bridge the gap between high-level application and low-level accelerators. Basically, such approach can be implemented based on Java Native Interface (JNI) [68]. Generally, the JNI is a native programming interface which allows Java code to inter-operate with applications and libraries written in other programming languages, for example C++.

Thus, we present a simple overview of FPGA-Adapter in Figure 4.2. As it can be seen, a JNI enables Java application which runs in its master thread, hands control to the native interface at the call site, stalls until the native thread returns and then continues execution.

The integration is done via function calls (of native code from main application). To simplify the development, we focus only on the Java-to-C++ function calls, since this

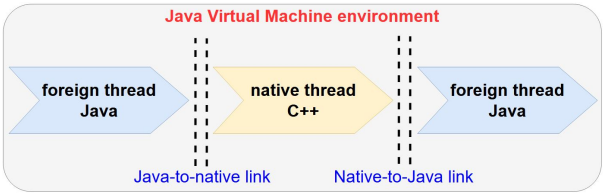


Figure 4.2 – Native Java-application overview.

yields the smallest overhead and allows full control of FPGA accelerators. The functional flow of deploying JNI can be described as follows:

— We implement the top-level architecture of MASCARA in Java, leaving out portions that are suitable for integration through JNI: legacy code or user implemented functionality.

— We create the native prototypes in the form of: *public native return_type nativeFunc(type1 arg1, type2 arg2,...);*

— We create JNI function headers to invoke FPGA accelerators through JDK tool (i.e., javah [69]). Then, we compile Java classes that contain native functions in the following form: *JNIEXPORT jobject JNICALL Java_SomeUserClass_nativeFunc(JNIEnv *, jobject, jobject, jobject,..);*

At high-level, we maintain the processing of MASCARA by object oriented, such as DNF-Object or CNF-Object meanwhile the kernels are developed functionally. Thus, FPGA-Adapter also handles the data representation (as show in Figure 4.3).

As depicted in Figure 4.3, a specialized interface, named Bean class, generalizes specification of computing modules in MASCARA, such as Intersection and Difference of predicate matching that are presented as objects of JavaType. Such implementation is derived from the JavaBeans concept presented by Java Enterprise Edition. In details, it is responsible to encapsulate the objects of MASCARA and their computing modules into native objects CppType with native fields, methods or constructors. Moreover, the Bean-derived user class should have a similar representation with accelerators, at least for the core components that are exposed both to the Java and C++ environment. Therefore, JavaType, CppType and their defined objects have the same fundamental data representation.

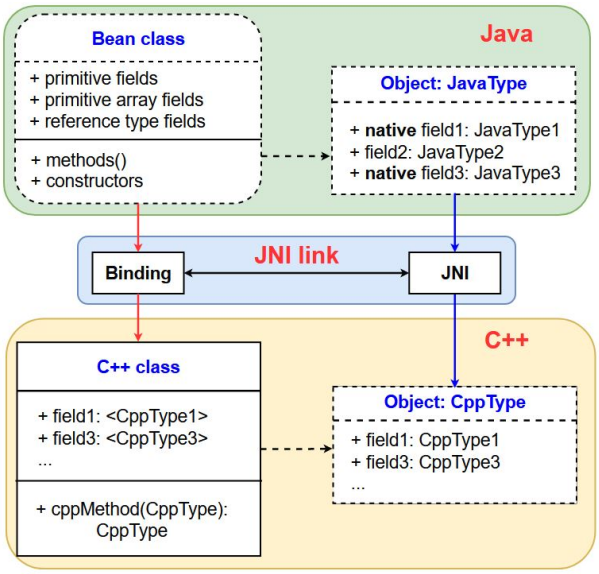


Figure 4.3 – Concept of data representation through FPGA-Adapter.

4.2.3 Accelerators of Query Trimming

Since the accelerators are used for computing modules of Query Trimming, we have three main categories: 1) Attribute Matching, 2) Predicate Matching, and 3) Semantic Extracting. In this dissertation, all of them are designed and described through a HLS language (i.e., C++14) within a particular platform (i.e., Xilinx Vitis HLS [53]). Thus, we present the details and functionalities of these accelerators with respect to a software-centric approach. Moreover, we describe their functions and tasks from bottom-to-top to emphasize the parallelism which can be achieved on FPGA. In particular, we first present the primitive computations that can be contained and used by the relevant top functions. Then, the top function will be wrapped into a kernel which belongs to one of the three above main categories. We illustrate such a kind of presentation in the following Figure 4.4. For example, to have the kernel $match\_Att\_ker$, we need to have the top function of this kernel: $match\_Att\_func$ which can consist of many primitive functions to complete its computation, such as $split\_element$, $common\_Att$ and $diff\_Att$.

Before presenting in detail the parallelism of Query Trimming accelerators, we present its data flow on FPGA (as shown in Figure 4.5). Considering that cache SC of MASCARA is organized on memory of FPGA (i.e., DRAM), to send the segments S to Semantic Matching's accelerators, we use two streams, one for the query and another for the list of segments. Note that the elements in the two streams could be pre-processed

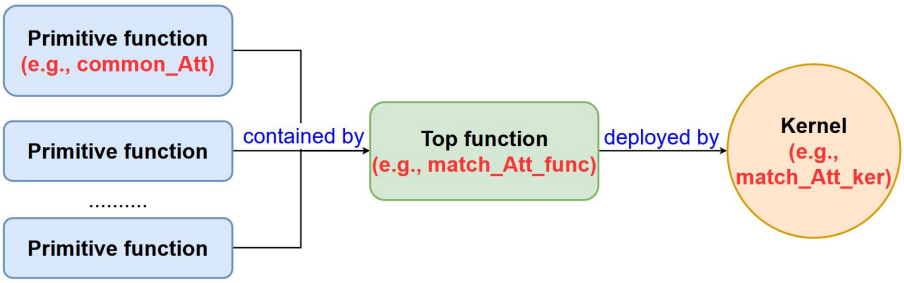


Figure 4.4 – An example of bottom-to-top presentation in the development of Query Trimming's accelerators.

to keep the necessary information (e.g, Q_{DNF}). Semantic Matching can run two accelerators in parallel, Attribute Matching and Predicate Matching. Moreover, to increase its throughput, multiple instances of these accelerators can be deployed to work independently. After matching, the outputs of the accelerators, such as an array of intersection predicates, or status of implication, could be stored temporally in on-chip memory (i.e., BRAM of FPGA) to be reused rapidly for next computations. By this way, they can be forwarded with a minimum latency to Semantic Extracting which is waiting for the signal from Semantic Matching (i.e., Predicate Matching) to be started. Since this accelerator contains mainly *if...else* condition statements to construct PQ and RQ, it does not require to divide into smaller specialized accelerated engines as in Semantic Matching. Similarly to Semantic Matching, we can also deploy multiple instances of Semantic Extracting.

a) Primitive functions

The basic elements of each accelerator are the primitive functions. Generally, they are the local functions invoked by the top function to run a computation. For example, the Intersection of Predicate Matching is a primitive function, named $inter\_dnf$. Interestingly, it can call other primitive functions in same context (i.e., Predicate Matching), such as *combine*, $isEqual$ and $isEmpty$ to process. Since the primitive functions have a similar design within HLS perspective, we focus here in the following $inter\_dnf$. Our choice is motivated by the fact that it can be seen as the basic computing of Predicate Matching, one of the most important accelerators in Query Trimming. Thus, the declaration of $inter\_dnf$ can be shown in HLS pseudo code as follows.

```
template <list_CNF List>
void inter_dnf (hls::stream<List>& query_strm,
                hls::stream<List>& segment_strm,
```

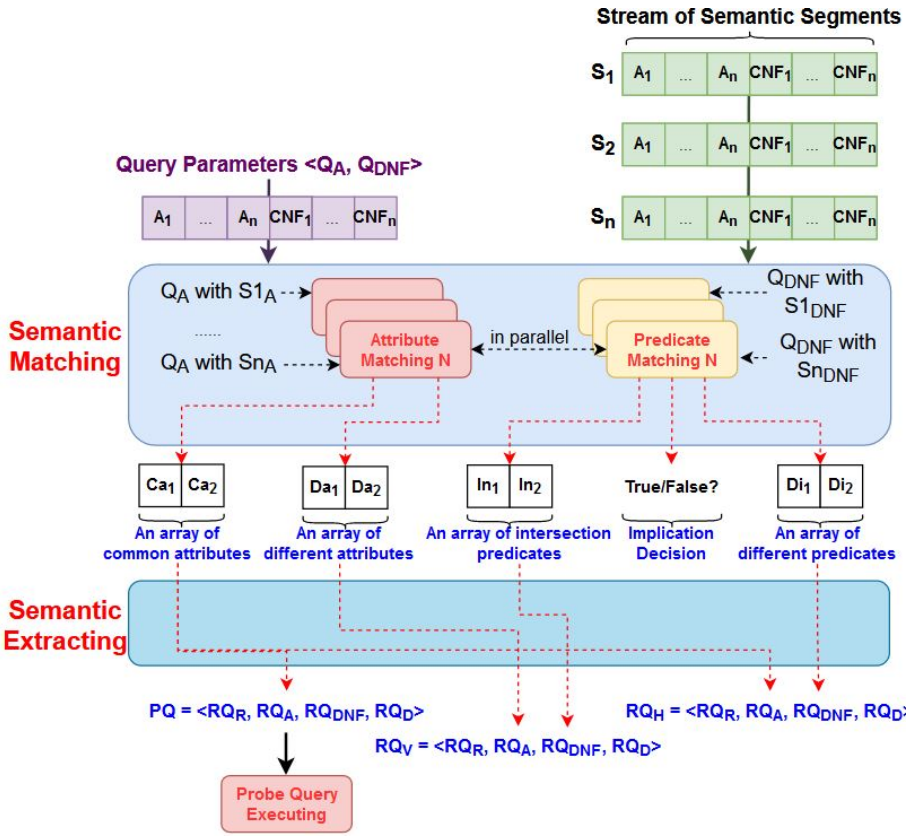


Figure 4.5 – Overview of the data flow in Query Trimming accelerators.

```
hls::stream<List>& result_strm)
```

As it can be seen, we create a streaming data structure based on predefined library *hls* :: *stream* to read the inputs from Q and S (i.e., *query_strm* and *segment_strm*). Moreover, outputs of the computation are put in to *result_strm* and can be transferred to next computation, for example, Implication *impli_dnf*. In order to explain in detail the advantage of *inter_dnf* on FPGA, we present an example as can be shown in Figure 4.6. Processing of tasks inside *inter_dnf* is the first level of acceleration that we gain compared to the baseline computing on CPU.

As depicted in Figure 4.6, we assume that this function consists of only three main operations: *operate1*, *operate2* and *operate3*. Note that *operate2* is implemented in terms of a nested loop to make the run over the lists of CNFs in Q and S. Regarding to MASCARA-Server, every operation only starts when the previous has finished since it can not be represented by a (logical) thread on the CPU. Indeed, manipulating operations or tasks to run concurrently in a thread can be seen as a challenge of engineering. As a result, this

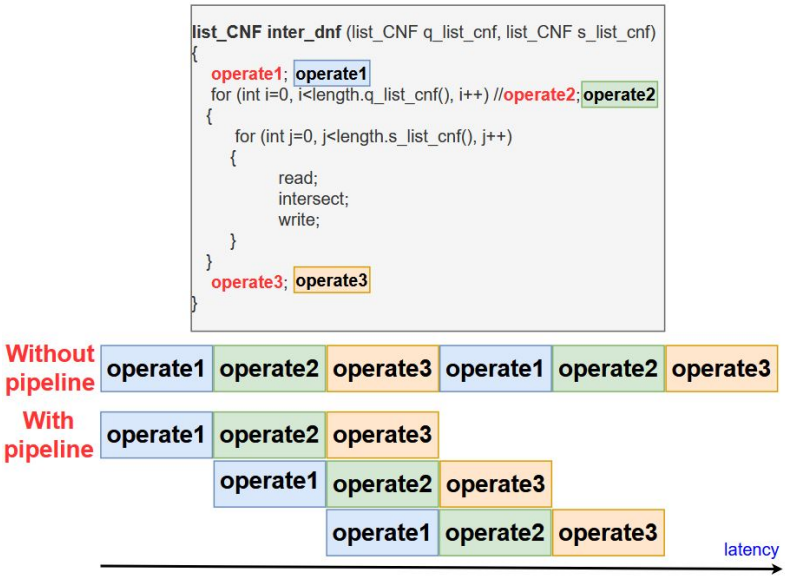


Figure 4.6 – Pipelining behavior for operations in function *inter_dnf*. Implementation by using a specified directive (i.e., *# pragma HLS pipeline*).

process on CPU requires a total of (N operations * *time_per_operation*) to complete all the computations of *inter_dnf*. In contrast, on FPGA, the operations of *inter_dnf* can be implemented to run in parallel in terms of pipeline execution model (as shown in Figure 4.6). Moreover, since these operators are decomposed and represented in a chain of simple arithmetic operators on FPGA, they can run rapidly without having to fetch, decode or execute like with the compute paradigm of CPU. All together, both the latency of execution and initiation interval between operations in *inter_dnf* can be improved substantially.

More importantly, *inter_dnf* has to iterate a nested loop over the lists of CNFs from two streams of Q and S that is presented in *operate*2. Let's assume that *operate*2 consists of three steps: *task1*, *task2* and *task3*. Thus, unrolling *operate*2 means that these steps can be executed in parallel (as can be shown in Figure 4.7). Based on several specific conditions, such as available hardware resources and data dependency between these steps, the unrolling can be done either partially or entirely on FPGA. In opposite, unrolling nested loop of *operate*2 on CPU raise a challenge of thread concurrency.

Thus, loop unrolling leads to a very fast design, with significant parallelism that can be seen as a complement of the pipeline for primitive function (e.g., *inter_dnf*). Nevertheless, the limitation of this approach is that we need a large amount of hardware resources. In particular, loop has a large number of internal operations and iterations.

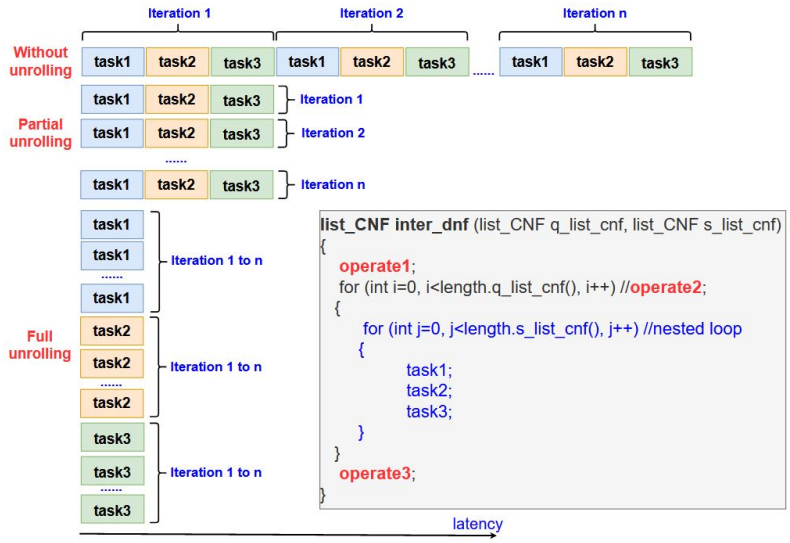


Figure 4.7 – Partially and fully unrolling nested loop (*operate*2) in function *inter_dnf* by using a specified directive (i.e., *# pragma HLS unroll*).

Regarding to MASCARA, the comparison between CNF-Objects can be done independently or we can say data dependency in this loop is zero. In other words, as long as we have enough hardware resources, the performance of *inter_dnf* could be maximized. However, MASCARA can be applied on different applications where we do not know exactly the number of CNF in Q and S. Thus, it is preferable to unroll the loop partially in MASCARA by defining a fixed value of unrolled iterations. As a consequence, by pipelining the execution of *inter_dnf* and unrolling its nested loop in MASCARA-FPGA, we achieve a task parallelism of primitive computing which can give better performance than MASCARA-Server.

b) Top functions

A top function, for example, *pre_top_func* can invoke the primitive functions, such as *inter_dnf*, *impli_dnf* and *diff_dnf*. Additionally, such kind of top function represent for the working perspective of accelerator, such as, attribute and predicate. In particular, *pre_top_func* is implemented on top of Predicate Matching accelerator. We illustrate the design of *pre_top_func* with inputs and outputs in following Figure 4.8. Remember that we can do similarly for other top functions, such as *att_top_func* and *ext_top_func*.

From two HLS streams of Q and S, *pre_top_func* distributes the arguments into relevant primitive functions, for example, *inter_dnf* or *impli_dnf* by defining its local

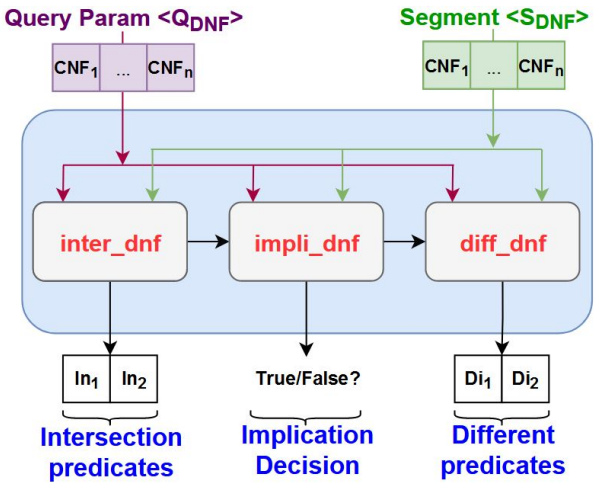


Figure 4.8 – Design of pre_top_func which is implemented on top of corresponding the Predicate Matching accelerator.

variables. Outputs of these computations are the elements to be reused soon in Semantic Extracting to generate the segments for PQ and RQ. Thus, they should be saved in BRAM to accelerate the decision making (i.e., relationship of matching) between Q and S. We remind that BRAMs is on-chip memory type with limited size which work as local and rapid storage for computations (as presented in Section 2.1.4). Fortunately, the outputs are presented in terms of arrays or lists of CNFs where each of them has a small size (i.e., less than $64bytes$). Meanwhile, size of the list depends on the number of dimensions and CNFs of Q and S. Practically, a list with up to hundreds of CNF-Objects only consumes a small amount of available BRAMs. For example, within a match between Q and S, with 50 CNFs for each, their results of Intersection and Difference consume less than $64KBytes$ that can be stored by two BRAMs. As a result, we can also store the outputs of Q and SC which consists of a list of S thanks to the sufficient size of BRAMs.

Similarly with the primitive functions, we also consider managing the workflow of pre_top_func to run its operations in parallel. Within an HLS approach, a directive can be specified in the implementation of pre_top_func. By this way, the HLS platform will analyze automatically the data flow between these tasks and create the appropriate channels that allow *consumer function* to start before the *producer function* completes More importantly, it requires us to apply the *producer-consumer* paradigm to pre_top_func in order to extract functionality that can be executed in parallel to improve performance. In other words, we have to figure out which parts of pre_top_func can be forked off for parallel computation and which parts need to be executed sequentially. As a result,

it can increase the overall throughput of *pre_top_func* in case data availability is high frequently. In fact, such approach can be done on CPU by creating a master thread which performs some initialization steps and then forks off a number of child threads to do some parallel computations. Although we can interleave the execution of the steps of each thread, this requires careful analysis to exploit the underlying multi-threading which is expensive in time.

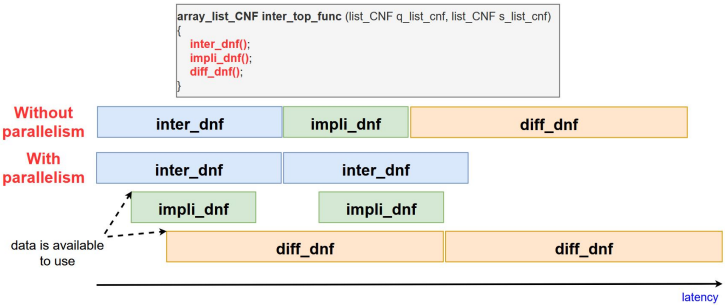


Figure 4.9 – Primitive functions of *pre_top_func* are pipelined as soon as data are available by using a specified directive (i.e., *# pragma HLS dataflow*).

We illustrate the execution in parallel of the primitives functions in *pre_top_func* in terms of data flow management through Figure 4.9. The default behavior is to execute and complete $inter\_dnf$, then $impli\_dnf$, and finally $diff\_dnf$. However, we can schedule the $impli\_dnf$ and $diff\_dnf$ to start as soon as the data (i.e., pair of CNF-Objects from query and segment) is available. For example, since the computations are iterated from CNF-Objects to Range-Objects, Difference between (Q_{CNF1}, S_{CNF1}) can start when Intersection between (Q_{CNF2}, S_{CNF2}) executes. To create data transferring channel between each of the functions, a first-in-first-out (FIFO) buffer acts as a queue to provide data-level synchronization between the functions and achieves better performance. The reason is that data stored in the array (i.e., a list of CNF-Objects) is consumed or produced in a sequential manner. Finally, due to the custom architecture of FPGA, these functions can be executed simultaneously with little or no overhead leading to a considerable improvement in throughput.

c) Accelerators

To use a top function (e.g., *pre_top_func*), it is essential to provide the relevant accelerator, in particular, Predicate Matching of Query Trimming. To maximize the performance, the accelerators of Query Trimming should run in parallel, taking into account

data dependencies (as shown in Figure 4.10). In particular, Query Trimming of MASCARA consists of three accelerators: Attribute Matching, Predicate Matching and Semantic Extracting. We consider that Attribute Matching and Predicate Matching can run in parallel since they work independently with two different perspectives. Meanwhile, Semantic Extracting has to wait and be invoked after Predicate Matching finishes.

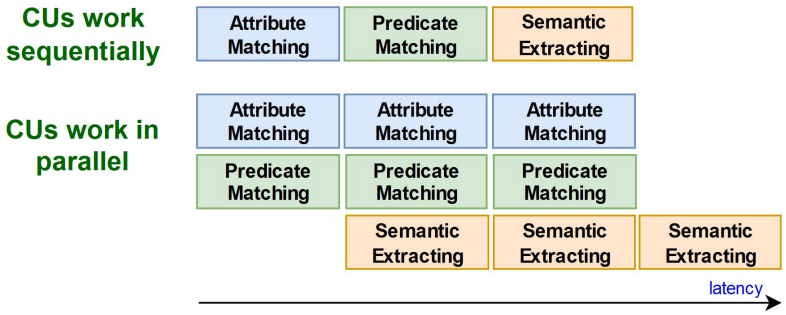


Figure 4.10 – Accelerators work in parallel. Each of them can be initialized in terms of one instance, called Computing Unit (CU).

To create such a kind of parallelism, we first deploy each accelerator in terms of a Computing Units (CU) that are identical clones. In fact, the number of generated CUs depends on the available hardware resources of the FPGA, such as Flip Flop (FF), BRAM, or Configurable Logic Block (CLB). It is essential to create a command queue to manage the order of execution as well as data flow for these CUs. Importantly, such queue has to keep the CUs as busy as possible to maximize the performance. Moreover, it is responsible to manage data dependencies between CUs through events and wait list. More precisely, if a CU depends on another one, the event should be passed into a wait list where all events have to finish before executing it. Thus, we can categorize the queue with two types: an in-order command queue and an out-of-order command queue.

Figure 4.11 illustrates how we use command queues to manage the order of execution between the accelerators. As it can be seen, we can do that by either using an out-of-order queue or two in-order queues. For the first approach, we have to define event dependencies and synchronizations explicitly. Meanwhile, the second approach pulls out the CUs in-order implicitly. In this dissertation, we use out-of-order command queue since we create more of them to manage multiple generated CUs of each accelerator later.

Obviously, with only one CU for an accelerator, MASCARA uses it in a sequential manner which can impact overall application performance. In particular, Predicate Matching with only one CU does not allow to run in parallel the matching of two pairs

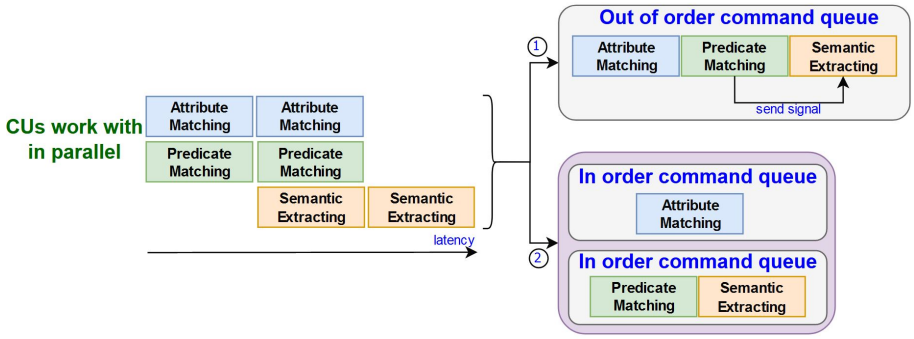


Figure 4.11 – Using an out-of-order command queue or two in-order command queues to manage the execution of CUs in parallel.

(Q_{DNF}, $S1_{DNF}$) and (Q_{DNF}, $S2_{DNF}$). Additionally, a high latency of Predicate Matching for $(Q_{DNF}, S_{i_{DNF}})$ increases the stall of comparing $(Q_{DNF}, S_{j_{DNF}})$ with $j = i + 1$ and $j < Nb\_Seg$ (number of segments) in SC. Fortunately, this issue can be overcome by instantiating multiple independent CUs from a single accelerator. By this way, the Predicate Matching composed by multiple CUs can execute in parallel. We illustrate the running of multiple CUs in parallel in Figure 4.12. In complement, out-of-order command queue is preferable to use in this case study since it avoids to manage a large number of in-order queues.

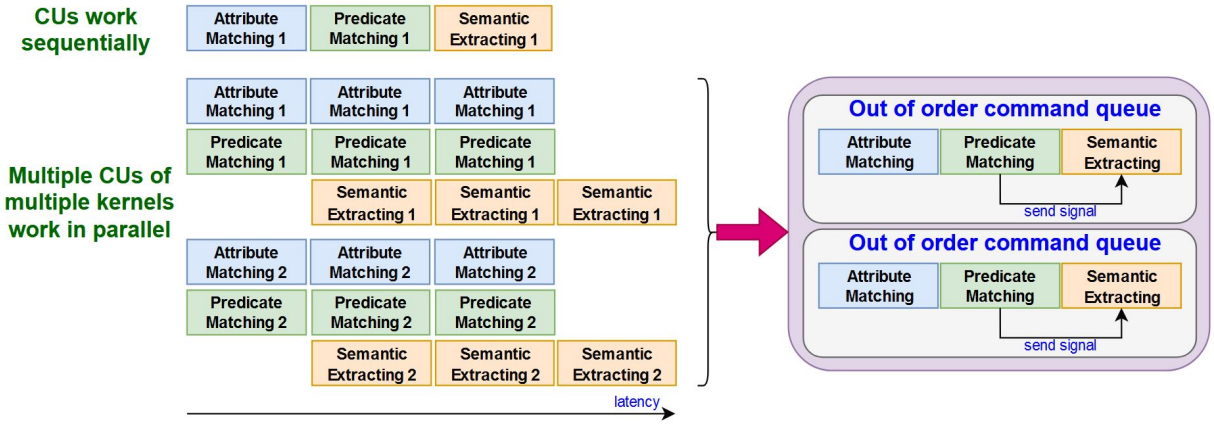


Figure 4.12 – Multiple of CUs (i.e., 2 CUs for each kernel) work in parallel. Two out-of-order command queues are instantiated to manage their data flow.

In conclusion, by designing Query Trimming's accelerators with respect to the bottom-to-top parallelism, the performance of MASCARA can be improved compared to the current implementation with multi-threads on CPU. In fact, due to the challenge of engineering, our implemented (logical) multi-threading over CPU with multi-cores cannot

support the parallelism of the tasks. In particular, the management of a master thread and its child threads can lead to the complex issue of thread concurrency. Moreover, sharing and moving data between threads are another problems that result to a challenge in memory management and synchronization. Meanwhile, FPGA facilitates the development of parallel execution in Query Trimming as we presented from primitive functions to accelerators in this section. Therefore, we consider that converting Query Trimming of MASCARA to FPGA accelerators is meaningful enough, especially when we can also leverage other benefits from FPGA, such as execution of PQ, Multi-view processing and coalescing heuristic.

Therefore, we consider Query Trimming is more suitable to be accelerated by FPGA which can boost up the performance of MASCARA as much as possible, especially with multi-dimensional condition.

4.2.4 Semantic Cache Management on FPGA

Since Query Trimming's accelerators takes the inputs (i.e., list of segments) to process, it is reasonable to move cache organization from the main memory of the server to the off-chip memory (i.e., DRAM) of the FPGA. Indeed, placing cache on FPGA side can reduce the communication with server's memory for each matching between the query and the list of segments. Although fetching data of DRAM is less efficient than BRAM, it has a larger space (i.e., up to 64GB with four DDR in Xilinx Alveo U200), which is ready to be run in big data applications. Combining the hierarchy of FPGA memory to deploy cache is not studied currently in this dissertation. Therefore, similarly with MASCARA on server node, MASCARA-FPGA contains two parts for semantic caching in DRAM: an index table that manages list of segments and a collection of corresponding data regions.

Using HLS development platform, we can declare a segment S on FPGA as a global variable (i.e., a template). Thus, S consists of the essential elements that are fully presented in Section 3.3.4. Moreover, lists of semantic segments S are composed of fixed size arrays in terms of Semantic Description Tables of SC. In fact, the maximum capacity of these arrays can be pre-defined statically before running MASCARA on FPGA.

Meanwhile, to create corresponding data region DR of S, we need to propose a mechanism of generating tables on FPGA. In particular, after executing the RQ on DMS, the result set is returned and stored in the main memory (i.e., RAM) of the compute layer. Since we have to cache this result on FPGA, MASCARA has to prepare a data structure that supports either generating row or column-oriented table on DRAM of FPGA. Thus,

a class or an interface which consists of the properties and methods to interact over these structures should be proposed. Obviously, this interface has to work with a well-known schema of the database (e.g., data sets of TPC-H [1]) where types of column are possible to be defined through the HLS platform. Based on this knowledge, a data structure can be created dynamically thanks to the arguments that represent existed attributes in the result set. Making a general interface for any kind of data sets has not been considered yet in this dissertation. All the above details can be summarized into Figure 4.13.

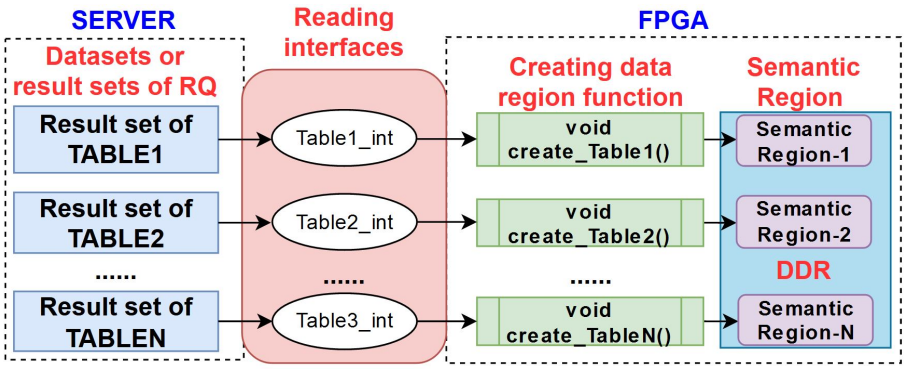


Figure 4.13 – A reading interface to create data regions on DRAM of FPGA.

As an example, we assume that the queries run over only one table (i.e., *lineitem*) generated from the data set of the TPC-H [1]. Within reading interface, we can create a class which represents row-oriented content of this table or part of it. Note that we have to define earlier several specified data types, such as *d_long* and *d_string* are wrapper of "long" and "char[]", respectively. This class consists of two main methods that overload the operators << and >> for input and output streams, respectively. A simplified version of this class is presented as following.

```
class lineitem_t { //Row oriented table
    public:
       d_long orderkey;
       d_long partkey;
       d_long suppkey;
       d_long linenumber;
       d_string<TPCH_READ_MAXAGG_LEN + 1> shipinstruct;
       //...other required attributes
       std::ostream& operator<<(std::ostream& output, lineitem_t& source);
       std::istream& operator>>(std::istream& input, lineitem_t& target);
```

Meanwhile, we also have another class for column-oriented table. Basically, it is similar with row-oriented class, except that the attributes are addressed through a vector. For example, *linenumber* is declared with *std::vector<d_long> linenumber* with the capability to be resized if necessary. Based on the reading interface, the server will create the corresponding buffers with required attributes and extra information, such as pointer of corresponding segment to transfer data to DRAM. From now, as we mentioned, it is pretty straightforward to be implemented as others read and write procedure between CPU and DRAM of FPGA. To optimize the space of data regions on cache, it is essential to know well about width and length of prepared regions on FPGA. A part of this procedure (i.e., creating buffer on server) can be described as following.

```
int create_buffers(cl_context ctx,
      cl_kernel kernel,
      int i,
      uint32_t* raw_filter_cfg,
      DATE_T* col_l_shipdate,
      MONEY_T* col_l_discount,
      ...
      cl_mem* buf_filter_cfg,
      cl_mem* buf_l_shipdate,
)
```

4.2.5 Accelerators of Probe Query

Besides Query Trimming accelerators, MASCARA-FPGA is responsible to execute probe queries (i.e., select-project-join query) based on the database (DB) operators. In the following sections, we present the principals as well as primitive functions which correspond to the steps in query execution plan. It is worth to note that the DB operators are developed similarly with Query Trimming accelerators with task parallelism from bottom-to-top.

a) Filter

Among the various operations in relational database, condition filtering or predicate evaluating is one of the most commonly used. More precisely, it can select the rows of interest from a table (relation) by applying filtering criteria (predicates) on one or more columns of the rows. From a CPU consumption perspective, this procedure becomes an

intensive operation as the number of predicates increases, especially for a large filtering from millions to billions of rows in big data context. Moreover, regarding with semantic caching (SC), a large number of generated probe queries PQ from Query Trimming, results to worse performance. Interestingly, we can represent such kind of filter through corresponding FPGA accelerators. In particular, since this expression-based filter works with Boolean conditions, in MASCARA-FPGA, we present an accelerator $filter\_kernel$ in which the expressions are evaluated in parallel (as shown in Figure 4.14).

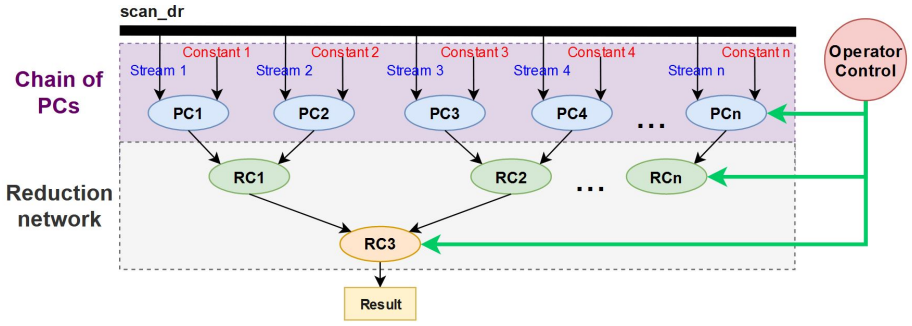


Figure 4.14 – Expression-based filter kernel. Predicate Cells (PCs) operate concurrently and independently. Reduction Cells (RCs) combine the individual outputs of PCs.

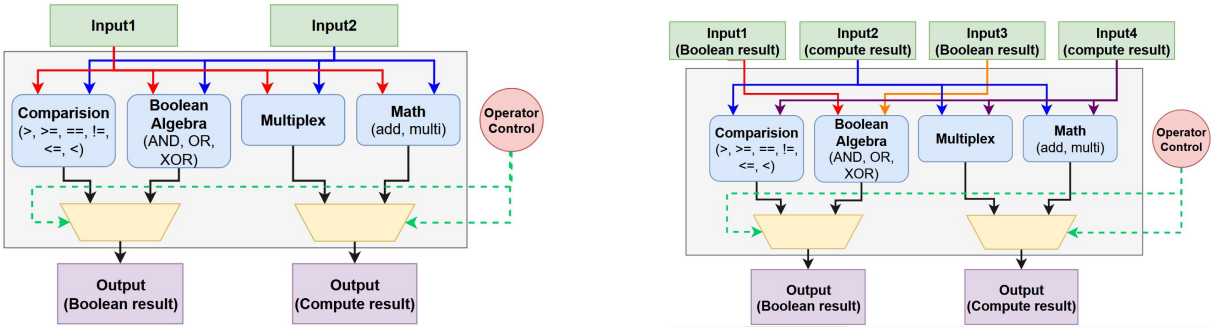
The basic unit (top function) for evaluating query predicates on FPGA is a predicate cell (PC) which receives data (rows) from HLS streams. We remind that they are used to transfer data from DDR to FPGA kernel and is contained in a primitive function, for example, $scan\_dr$ for scanning data region as following. Then, a PC evaluates an atomic predicate by comparing a constant value (c) against up to a 64 bits long column of the streaming database rows. A chain of PC(s) (i.e., $PC1 - PC4$) can be organized to evaluate complex predicate expressions since they can run concurrently and independently each other. Later, a network of Reduction Cells (RCs) (i.e., $RC1 - RC3$) combines the individual outputs of the PCs. In other words, this network works sequentially with the PCs. Since PCs and RCs consists of different computation modules, an operator control is essential to indicate which operation they should process.

The number of PCs and the size of the RC network inside a filter kernel are variable which results to a trade-off between area and complexity. For example, our $filter\_kernel$ is declared with four streams and four constant values as inputs. However, it can be extended with more input streams (e.g., six streams) to achieve a faster evaluating. Note that, for certain queries, the number of PCs inside the scanner may be fewer than that required to evaluate the query. In addition to the predicate level parallelism, $filter\_kernel$

of MASCARA-FPGA also provides row level parallelism by creating multiple instances of *scan_dr*.

We illustrate the internal structure of PC in *filter_kernel* by Figure 4.15a. As it can be seen, it can provide up to four kinds of computing unit: Comparison, Boolean algebra, Multiplex and Math. For the first two computations, the results can be expressed in Boolean, while the two last can generate non-Boolean results. Meanwhile, RC needs to process both Boolean and non-Boolean results from PCs (as shown in Figure 4.15b). In particular, based on basic operators, such as AND, OR, NOT, a RC can implement switch-case structure in a comparison module to make a Boolean algebra between the inputs.

Assuming that to perform *date < '2022-11-11'*, $PC1$ takes first input *date = '2022-10-11'* as a record of the data set. The second (constant) input is *'2022-11-11'*. Obviously, passing the comparison module, the output is $false$. Meanwhile, at the same time, we also need to perform *ip > '192.168.1.100'* on $PC2$. Consider that the output of this comparison is *true* with first input *ip=192.168.1.111*. Finally, on $RC1$, it makes a decision based on Boolean algebra (e.g., AND, OR) for two inputs: $false$ and *true*. As a consequence, if the output is *true*, the input row is satisfied with condition and vice versa when output is $false$.



(a) Predicate Cell (PC). (b) Reduction Cell (RC).

Figure 4.15 – Internal structure of PC and RC.

b) Project

Projection is the operation that extracts desired attribute fields (columns) from a database row based on "SELECT" statement. Generally, it can be seen as the complement of the filter operation. On MASCARA-FPGA, we present the project kernel which can

perform in parallel filters and providing bandwidth and storage savings. Indeed, this result is achieved by removing unwanted columns from data regions which need to be cached with respect to the Query Trimming of MASCARA. Moreover, this kernel is essential to extract the dedicated columns that form the key for sorting the records, making projection a prerequisite step for the sort operation of merge join later.

We illustrate the integration of project and filter kernel in Figure 4.16 in the case of query processing in pipeline. Obviously, each filter is paired with a project kernel. Rows are evaluated against the predicates in the filter, then projected in the project kernel which captures the required columns and forwards them to the projected row buffer and/or the sort key buffer.

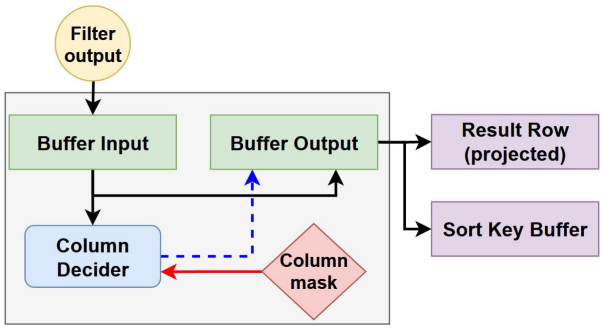


Figure 4.16 – Internal structure of Project which selects the projected columns from the output of Filter kernel.

In details, it consists of a module called Column Decider which masks the selected columns based on one-hot encoding (i.e., each bit corresponds to a column name). Then, if the corresponding bit for a column name is one, data field of this column is put into the output buffer (e.g., projected row buffer). Otherwise, it is discarded. To simplify the implementation, we consider that the columns have a fixed length that can be predefined before processing the query. For example, it can operate directly on the incoming 64 bits streams that conveniently correspond to the data width of a column. As consequence, it is worth to note that the implementation of project can be optimized by merging filter kernel. In other words, we could have a two step filter-project accelerator for both operations: pre-fetching and processing for filter to improve the throughput of the query execution.

c) Sort-Merge

A join operation aims to match records from two or more tables based on a common field (i.e., the key field). Since the nested loop implementation increases computational

complexity, most DBMSs deploy sort-merge-join or hash-join for efficient computation. Hash-join often performs better because of its linear algorithmic complexity, though hashing introduces false positives that must be resolved. An another commonly applied method for joins is sort-merge join where the main operation is sorting the tables to be joined. In this dissertation, we choose to implement sort-merge-join to support processing inner join query regarding to Query Trimming of MASCARA.

As the first stage in sort-merge-join, we have to consider a variety of sorting algorithms that have different impacts to hardware resource requirements, throughput (i.e., number of keys sorted in a given time), sorted run size (i.e., number of keys to be sorted) and the size of the sort key. Several works have implemented a single sorting algorithm on FPGA, meanwhile some have explored high performance sorting large size inputs with low level hardware description languages [36]. To simplify the development, in this dissertation, we focus only to the Merge Sort algorithm (for primitive function) within the tree structure called Merge Sort Tree (for top function). To explain, merge sort is organized in a strict data flow and there is no need for any control logic. Other sorting algorithms or networks, for example bitonic network, could be considered as an extension of MASCARA-FPGA in future.

The divide-and-conquer procedure of sorting (by joined key) an input (i.e., result table of sub-query in Multi-view processing) can be described as following:

— Dividing stage: we divide this unsorted input into n sub-inputs by their joined key. Each sub-input contains one element, thus it is considered sorted.

— Conquering stage: we merge repeatedly sub-inputs to produce new sorted sub-output until there is only one sorted sub-output remaining at the end.

On MASCARA-FPGA, regarding the conquering stage, we present a merge primitive called Merge Unit (MU) to combine two sorted $n/2$ size inputs into a sorted output of size n. We illustrate the structure of MU in Figure 4.17. The comparison component is implemented with processing elements that can sort when two conditions are met: the output is not full and both input hold data to sort. If these conditions are met, sorting takes place from the input and the smaller value will be pushed into the output.

This primitive function can be described by the following HLS simplified code. Note that its implementation works with the streaming First In First Out (FIFO) inputs and output. As it can be seen, $IN1$ and $IN2$ are two sorted arrays and OUT is the merged outputs. The for loop runs n times where $n/2$ is the size of IN1 and IN2. It reads one elements from either $IN1$ or $IN2$ on each iteration and writes it to the output until the

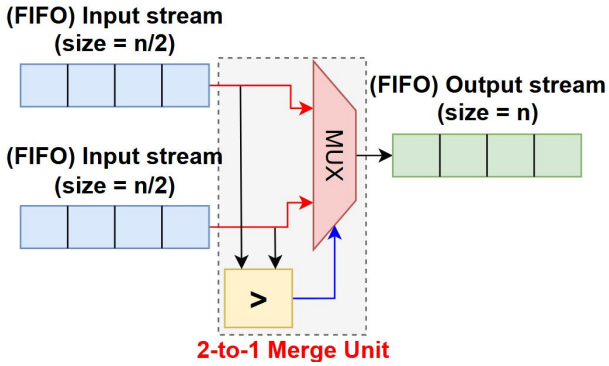


Figure 4.17 – Internal structure of 2-to-1 Merge Unit.

end of the FIFO is reached. The tasks of this loop is also pipelined to operate one read at every cycle.

```
void merge_Unit(hls::stream<int> &IN1, hls::stream<int> &IN2
                hls::stream<int> &OUT, int n)
{
    int a, b;
    int index1 = 1, index2 = 1;
    IN1.read(a); IN2.read(b);
    for (int i = 0; i < n; i++){
        if(index1 == n/2 + 1){
            OUT[i] = b;
            IN2.read(b);
            index2++;
        } else if(index2 == n/2 + 1){
            //... }
        else if (a < b){
           //... }
        else {
           //... } //end of if-else
    } //end of for
}
```

Generally, from pure software perspective, this primitive function will be called recursively to address the divide-and-conquer procedure. However, recursive function calls is no longer allowed on HLS tools (i.e., Xilinx Vitis HLS [53]). Thus, to overcome this issue and employ multiple levels of MUs simultaneously, we organize them in terms of a

binary logical sort tree structure (as shown in Figure 4.18). In this example, there are eight pre-sorted inputs and each of them has a size of $n/8$. Thus, we can deploy a tree of seven MUs in total to merge their results together through three stages from top to bottom. By this way, such kind of tree can replace a recursive function with a higher parallelism thanks to multiple MUs. In general, for a depth n merge sort tree, it requires $O(2^n)$ FIFO entries that can consume a large amount of memory resources. In short, merge sort tree on MASCARA-FPGA can run $nlog(n)$ tasks in parallel to achieve $O(n)$ complexity. However, the area consumption grows exponentially with the problem of size. Note that for large size n, the data buffering process in the node (MU) at the bottom of the tree has to be performed using device memory (DDR) which thus could impact the performance. Furthermore, if the size of the tree is small compared to unsorted input, we have to split this input to smaller parts and then call the tree multiple times to sort. As a result, this problem damages greatly the response time of sort merge on FPGA.

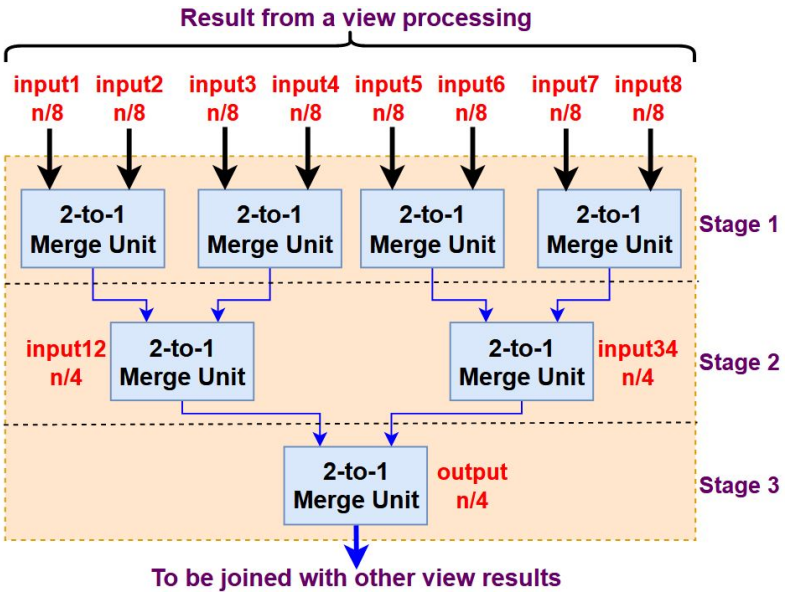


Figure 4.18 – Sort merge tree structure with depth=3.

d) Pre-sorted join

After having the results from multiple views that are sorted by presented merge-sort tree, the last step is to join them together based on their joined condition (i.e., joined key). We illustrate the design of presorted-join operator in Figure 4.19.

The top function *merge_Join* receives two FIFO input streams that consist of the

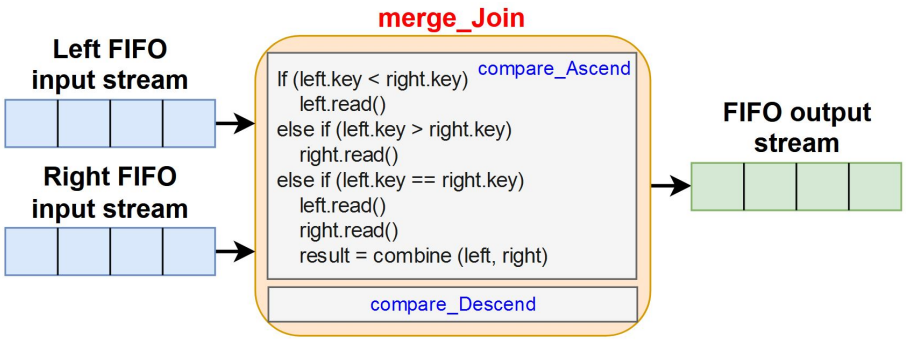


Figure 4.19 – Join of two ascend tables. Descend tables can be done similarly.

key type and payload (content) type. In other words, the left and right result tables of views are pushed out in separate streams. Assuming that we need to join two tables in ascending order, every clock cycle, *merge_Join* compares the keys from left and right tables by primitive function *compare_Ascend*. If the keys are not the same, it pulls the stream with a smaller key and there is no output. If the keys are the same, it pulls the right stream and pushes the keys and payloads to the output stream.

4.3 Validation

In this section, we present the experimental results with the following objectives. First, we validate the performance of MASCARA-FPGA compared to other solutions, such as MASCARA-Server, Block-Cache and No-Cache. Second, we show the acceleration in response time of Query Trimming and Probe Query. Third, we exhibit that the bottleneck of Multi-view processing can now be handled by MASCARA-FPGA thanks to its top-to-bottom parallelism. Fourth, we reevaluate the benefits of coalescing heuristic on MASCARA-FPGA.

4.3.1 Validation environment

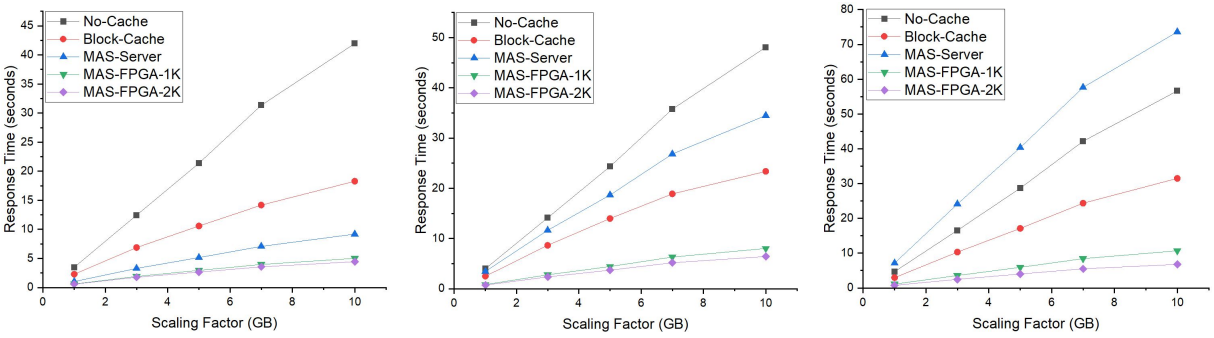
The experiments to evaluate MASCARA-FPGA have the same environment set up as presented for MASCARA-Server in Section 3.4.1. In particular, data sets, workloads, warm-up queries, measurement method, evaluated metrics, impact factors and testing prototypes are reused. Moreover, since MASCARA-FPGA can deploy multiple kernels to run in parallel, we use two new testing prototypes: MAS-FPGA-1K as using one kernel and MAS-FPGA-2K as using two kernels.

Moreover, to run MASCARA-FPGA, we need to use an accelerator card as an instance of the FPGA side. Thus, we use Xilinx® Alveo™ U200 Data Center accelerator card which has 65MB Block RAM and four 16GB off-chip memory DDR4 with total bandwidth up to 77GB/s. The connection between CPU (server node) and FPGA is established by PCI Express Gen3x16 and network interfaces typed Ethernet support 100Gbits/s.

4.3.2 Experimental results

a) Overall performance

Performance of Q6 workload. We present the following Figure 4.20 in which we change the dimension of query Dim_Query and SF since they can affect significantly the performance of MASCARA. To simplify the presentation of results, we only exhibit the most complex case where SC has 250 segments.



(a) $Dim\_Query \leq 3$ (b) $3 < Dim\_Query \leq 5$ (c) $5 < Dim\_Query \leq 8$

Figure 4.20 – Response times of Q6 mod

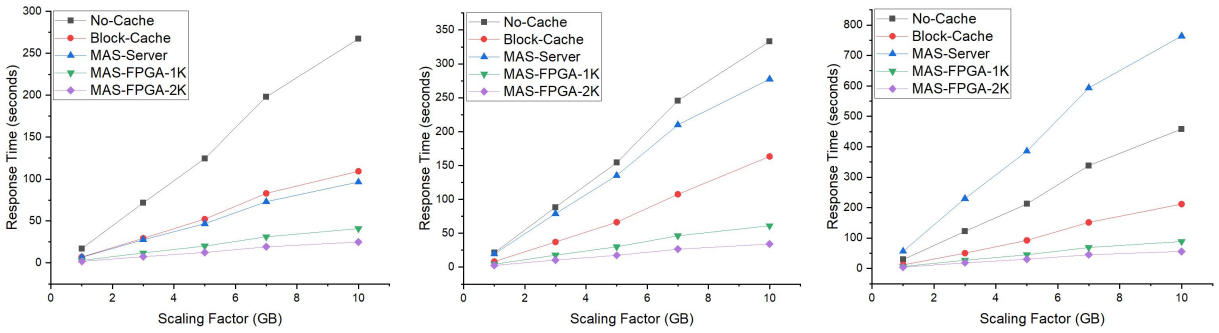
In Figure 4.20, obviously, response times of No-Cache are the highest for all SF as discussed in Section 3.4 since all queries are executed by DMS on server node. Meanwhile, MAS-Server and Block-Cache can reduce the response time. However, when number of dimensions in Q and S is high, MAS-Server is less efficient than Block-Cache (as shown in Figure 4.20b and even worse than No-Cache (as shown in Figure 4.20c) due to the overhead of Query Trimming.

With MAS-FPGA, the performance of MASCARA is now guaranteed. In Figure 4.20a, with $SF = 1GB$, MAS-FPGA-1K is 5., 3.8 and 1.7 times faster than No-Cache, Block-Cache and MAS-Server, respectively. Interestingly, unlike MAS-Server where the performance declines in case of having a high number of dimensions, MAS-FPGA overcomes

such kind of bottleneck. For example, in Figure 4.20c, with $SF = 1GB$, while MAS-Server is even 1.5 times slower than No-Cache, MAS-FPGA-1K can achieve up to 4.2 and 2.7 times faster than No-Cache and Block-Cache, respectively. In other words, MAS-FPGA-1K accelerates 6.4 times MAS-Server's processing. Thus, we acknowledge that applying MAS-FPGA can lead to better results in acceleration, especially when MAS-Server is no longer able to handle Query Trimming with high complexity (i.e., $5 < Dim \leq 8$ and $Nb\_Seg = 250$). Indeed, without parallelism of tasks or data processing in threads, MAS-Server runs the nested loop (i.e., intersection or difference) inefficiently. In opposite, MAS-FPGA-1K can run in-parallel from tasks to top functions, thus, it alleviate this issue to get stable response times with minimum effects from multi-dimensions.

Moreover, with two kernels deployed, MAS-FPGA can achieve a better performance. For example, with $SF = 1GB$, MAS-FPGA-2K is 6.4, 4.0 and 9.7 times faster than No-Cache, Block-Cache and MAS-Server, respectively (as shown in Figure 4.20c). It is worth noting that having two kernels does not mean the acceleration can be doubled since the output results also depend on the optimization of the workflow in processing of MASCARA-FPGA. For example, multiple mixed matching needs to check not only the (Q,SC) but also the (RQ,SC) that can reduce the benefits of multiple kernels.

Performance of Q5 workload. We present the Figure 4.21 to prove that MASCARA-FPGA can overcome the issue of Multi-view processing which happens in execution of inner join queries (i.e., workload of Q5).



(a) $Dim\_Query \leq 3$ (b) $3 < Dim\_Query \leq 5$ (c) $5 < Dim\_Query \leq 8$

Figure 4.21 – Response times of Q5 mod

As it can be seen, we reconfirm that although MAS-Server enables Multi-view processing for workload of Q5(s) customized, its performances decline dramatically when the number of dimensions is high. Moreover, this decline is more severe than it is in workload

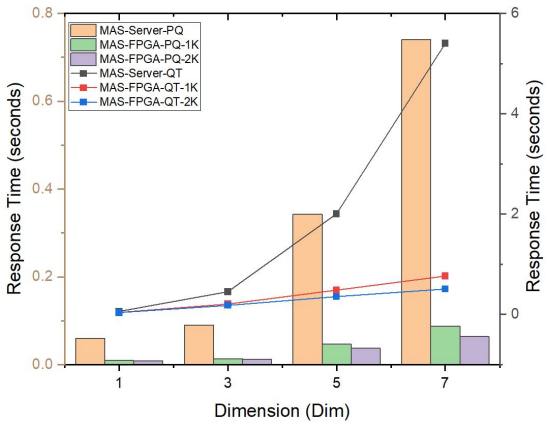
of Q6(s). The main reason is invoking Query Trimming multiple times and sort-merge-join is not optimal on MAS-Server. More details have been already presented in Section 3.4. Meanwhile, MAS-FPGA which has accelerators for Query Trimming and sort-merge-join, can overcome the bottleneck of Multi-view processing. For example, in Figure 4.21c, with $SF = 1GB$ while MAS-Server is even 1.9 times slower than No-Cache, MAS-FPGA-1K can be 4.4 and 1.8 times faster than No-Cache and Block-Cache, respectively. In other words, MAS-FPGA-1K accelerates 8.5 times MAS-Server's processing. Interestingly, we acknowledge the impact of accelerators in MAS-FPGA-1K which become more remarkable compared to both No-Cache and MAS-Server when SF increases. In particular, with $SF = 10GB$, MAS-FPGA-1K is 5.2, 2.4 and 8.7 times faster than No-Cache, Block-Cache and MAS-Server, respectively.

Similarly with workload of Q6(s), we also see an acceleration of MAS-FPGA when using two kernels. Indeed, we increase the performance of Multi-view processing by handling two (i.e., *lineitem* and *orders*) out of three views in parallel before joining their viewed results at the end. In particular, with $SF = 1GB$ and dimension changes from simple (i.e., $Dim \leq 3$) to complex (i.e., $Dim \leq 8$) (as shown in Figure 4.21), MAS-FPGA-2K brings an acceleration up to 3.9, 8.1 and 13.7 times compared to MAS-Server. Therefore, we can conclude that Multi-view processing is preferable to be deployed in MAS-FPGA, with multiple instances (i.e., Query Trimming) accelerators.

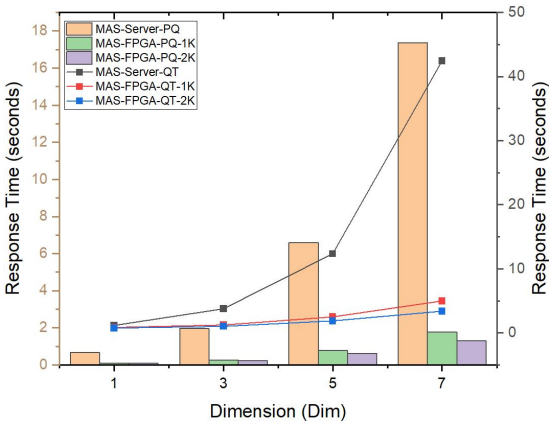
b) Performance of accelerators

In this section, we present in details the performance of Query Trimming and DB operators in terms of PQ execution. To simplify the presentation of the results, we focus only to MAS-Server and MAS-FPGA that have same computing modules. Using only one kernel is reasonable enough to present the impact of acceleration which can be gained by FPGA compared to server when changing the coalescing strategies. Moreover, we present the results of MASCARA with $SF = 1GB$.

Query Trimming. As it can be seen in Figure 4.22a with workload of customized Q6(s) and $SF = 1GB$, when Dim is small (i.e., $Dim < 3$), MAS-Server is capable to handle the computation in Query Trimming in a relative short time. For example, with $Dim = 1$, MAS-FPGA with one kernel of Query Trimming, named MAS-FPGA-QT-1K, is only 1.5 times faster than Query Trimming on MAS-Server (i.e., MAS-Server-QT). Using two kernels, MAS-FPGA-2K-QT increases slightly the performance, in particular, it is 1.6 times faster than MAS-Server-QT. The reason is that the second kernel is not used



(a) QT and PQ in workload of Q6(s).



(b) QT and PQ in workload of Q5(s).

Figure 4.22 – Execution time of Query Trimming (QT) and Probe Queries (PQs) with workload of customized Q6(s) and Q5(s) respectively.

since Query Trimming can be handle-well by one kernel. In other words, the performance of two kernels is not impressive with light computations in Query Trimming. In contrast, when number of dimensions increases (i.e., $Dim \geq 3$), accelerators have a real impact on performance. In particular, wit $Dim = 7$, MAS-FPGA-QT-1K is 7.1 times faster than MAS-Server-QT. Moreover, in case of two kernels that run with minimum idle state, this acceleration is increased to 10.7 times.

Similarly, in Figure 4.22b with workload of customized Q5(s), when number of dimensions is high (i.e., $Dim = 7$), MAS-FPGA-QT-1K is 8.5 times faster than MAS-Server. Since Query Trimming is called multiple times in this case, performance of MAS-Server is far behind that of MAS-FPGA-QT-1K. Cloning the kernel by two (i.e., MAS-FPGA-2K-QT) can boost up this acceleration to 13.2 times by processing multiple views in parallel. To summarize, Query Trimming on MAS-FPGA shows better performances compared to MAS-Server, especially when dimension of query is high.

Filter and Project. Another aspect of acceleration, which can be gained on FPGA, is executing generated select-project PQ. Figure 4.22a shows the response times of executing PQ which consists of filter-project accelerators. In particular, MAS-FPGA-PQ-1K is 6.3 and 8.5 times faster than MAS-Server-PQ, with $Dim = 3$ and $Dim = 7$, respectively. We notice that there is a slight increase of acceleration gained by MAS-FPGA-PQ-1K when number of dimensions is high. The reason is that the formats of generated PQs become more complex, thus, MAS-Server takes more time to execute meanwhile MAS-FPGA-PQ-1K has in parallel execution presented in filter-project accelerators. Moreover,

this is also related to data regions (DR) that have different sizes, such as number of attributes or records. Within these experiments, most of DR size are less than 3% of its related table(s) no matter what the dimension is. In fact, MAS-FPGA-PQ-1K has enough hardware resources to handle efficiently such volume of data for each generated PQ. Thus, the acceleration of PQ on MAS-FPGA-PQ-1K is guaranteed. Moreover, with two kernels, MAS-FPGA-PQ-2K can accelerate more, in particular, it is 11.5 times faster than MAS-Server-PQ.

Similarly, with workload of Q5(s) in Figure 4.22b, MAS-FPGA-1K-PQ is 9.8 times faster than MAS-Server-PQ. In this case, more numbers of PQs can be generated due to Multi-view processing. Thus, the acceleration of MAS-FPGA-1K-PQ on workload of Q5(s) is increased slightly compared to the acceleration on workload of Q6(s). Deploying two kernels (i.e., MAS-FPGA-PQ-2K), the acceleration can be increased up to 13.2 times compared to MAS-Server-PQ.

Sort-Merge-Join. We consider that the acceleration of sort-merge-join for the result (tables) from each view in MASCARA does not have any affect from dimensions in the query. However, with a fixed number of joined views (i.e., three relations), sort-merge-join depends on the scaling factor SF. Thus, we exhibit the performance of sort-merge-join in Figure 4.23 regarding to change of SF.

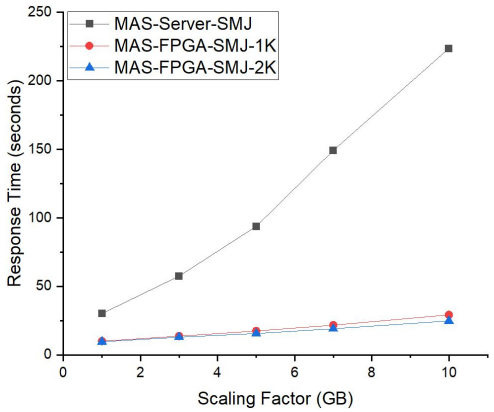


Figure 4.23 – Sort-Merge in workload of customized Q5(s).

On MAS-FPGA, join is implemented as a procedure of if-else statement which processes at high speed over two pre-sorted (on join columns) input streams. On the other hand, sort is deployed by a sort-merge binary tree which consists of 2-to-1 merge units (MUs) limited by the resources of FPGA and impacted by the size of the input. Thus, we consider that sort process is the main element to bring the acceleration of sort-merge-join

on FPGA.

With one kernel, MAS-FPGA-1K-SMJ is 3.0 times faster than MAS-Server-SMJ when data set is small ($SF = 1GB$). Moreover, with $SF = 10GB$, the acceleration can be up to 7.7 times. The reason is that execution time of MAS-FPGA-1K-SMJ increases slightly with its large sort-merge-tree meanwhile MAS-Server-SMJ runs slowly divide-and-conquer sorting. With two kernels, MAS-FPGA-2K-SMK can achieve an acceleration around 9.0 times compared to MAS-Server-SMJ with $SF = 10GB$. Importantly, such a kind of acceleration could be much more in case the number of views increases and each of views has a large content combined from multiple of PQ and RQ.

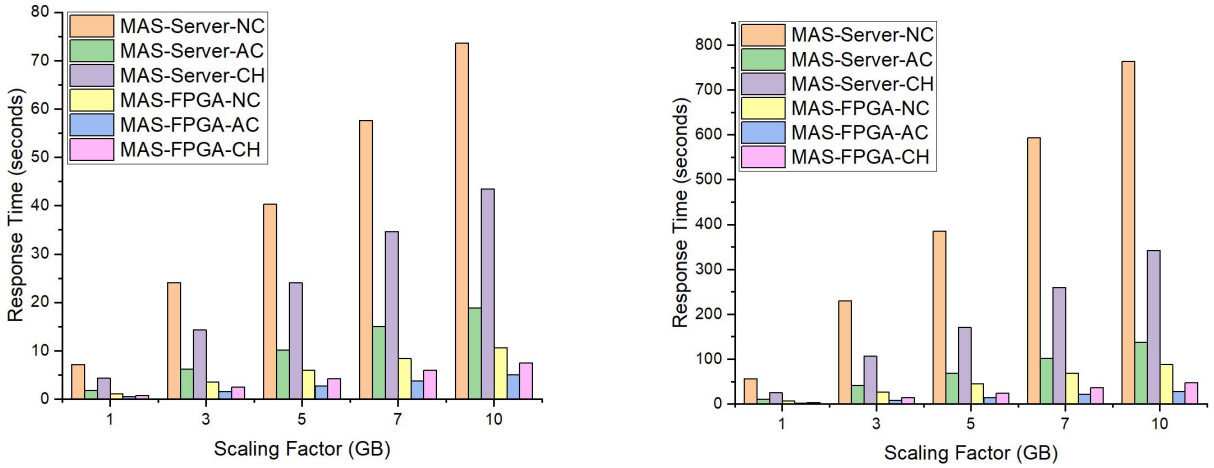
Theoretically, without considering memory capacity and bandwidth connections, we can say that there is no limit on how many 2-to-1 Merge Unit (MUs) could be deployed using such a sort-merge scheme. In other words, with sizes of results from which each view are frequently less than 5% of the relation, it is reasonable to have a sort merge tree up to 512 2-to-1 MUs. Moreover, to reduce the recursive complexity, this tree could be expressed as multiple sorting networks for un-sorted data before processing in each networks if there are enough hardware resources on FPGA. Therefore, unlike MAS-Server-SMJ, such kind of implementation keeps sort-merge-join to operate with low latency most of the time even if SF increases.

To summarize this section, although each of the presented accelerators can achieve impressive results compared to MAS-Server, their aggregated result can be limited by the frequency of RQ occurrence and their execution time. Moreover, in case of multiple computing nodes for RQ, this issue could be more severe due to restricted communication between compute and storage layers. As a result, the acceleration of MAS-FPGA could decline since it takes a small partition of total response time which includes RQ execution. In our experiment, hit ratio of MASCARA is guaranteed above 90% with respect to high semantic locality in the workloads and data sets that are presented by SK and HR. Therefore, MAS-FPGA maintain its high acceleration compared to MAS-Server. In particular, it is 6.9 and 8.7 times faster compared to MAS-Server with workload of customized Q6(s) and Q5(s), respectively.

c) Performances with coalescing heuristic

We remind that MASCARA based on CPU limits the benefits of two other contributions, coalescing heuristic for semantic management and multi-view processing for (inner) join query. Currently, with the presented accelerations, especially in Query Trimming,

with MASCARA-FPGA, we have to re-evaluate the benefits which can be gained by the coalescing heuristic. Thus, we present the results by two figures: 4.24a for workload of Q6(s) and 4.24b for workload of Q5(s). In short, Always Coalescing, Never Coalescing and Coalescing Heuristic are named respective AC, NC, and CH (with threshold $T = 0.5$) to be applied on MAS-Server or MAS-FPGA. Thus, we have a total of six testing prototypes.



(a) Workload of customized Q6s.

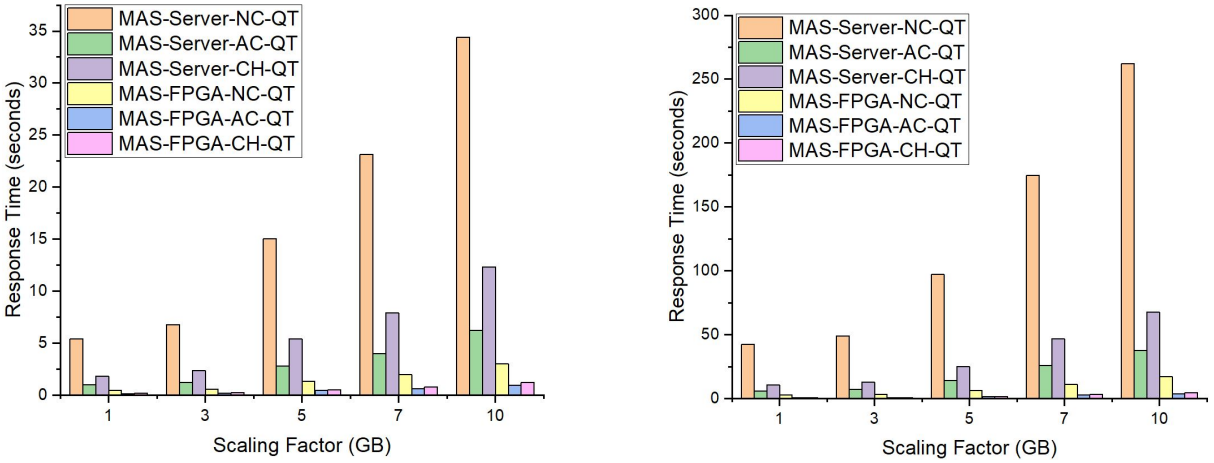
(b) Workload of customized Q5s.

Figure 4.24 – Response times of MASCARA with respect to different coalescing strategies.

Regarding to response time, Always Coalescing can bring the best results, coalescing heuristic shows a balance and Never Coalescing takes the last place for either server or FPGA. Considering that MAS-Server prefers to apply Always Coalescing since it can reduce the generated segments, thus resulting to better response times compared to Never Coalescing and Coalescing Heuristic. In Figure 4.24a, we found that the accelerations which can be gained within these pairs (MAS-FPGA-NC, MAS-Server-AC), (MAS-FPGA-CH, MAS-Server-AC), and (MAS-FPGA-AC, MAS-Server-AC) are on average 1.7, 2.5 and 3.8 times, respectively. Or in Figure 4.24b, they are 1.5, 2.85 and 4.9 when running workload of Q5(s). As it can be seen, despite the change of coalescing strategies, MAS-FPGA is always faster than (the best case of) MAS-Server. More importantly, we notice that there is a slightly decline in acceleration compared to MAS-Server when changing from Always Coalescing to Coalescing Heuristic on MAS-FPGA. On the other hand, we also notice that MAS-FPGA-AC is on average 1.5 times faster than MAS-FPGA-CH when running workload of Q6(s). Coordinating all the above comparisons, we consider performance of MAS-FPGA-CH is not too much inferior to MAS-FPGA-AC since its generated segments can be handled well within the capability of Query Trim-

ming's accelerators. Thus, changing from Always Coalescing to Coalescig Heuristic on MAS-FPGA allows to continue having an appropriate acceleration while also maximizing the other benefits of Coalescing Heuristic such as hit ratio and cache space usage.

Since Query Trimming is the most clearly affected when changing coalescing strategies, we also show its performance on FPGA and server to evaluate (in Figure 4.25).



(a) Workload of customized Q6s.

(b) Workload of customized Q5s.

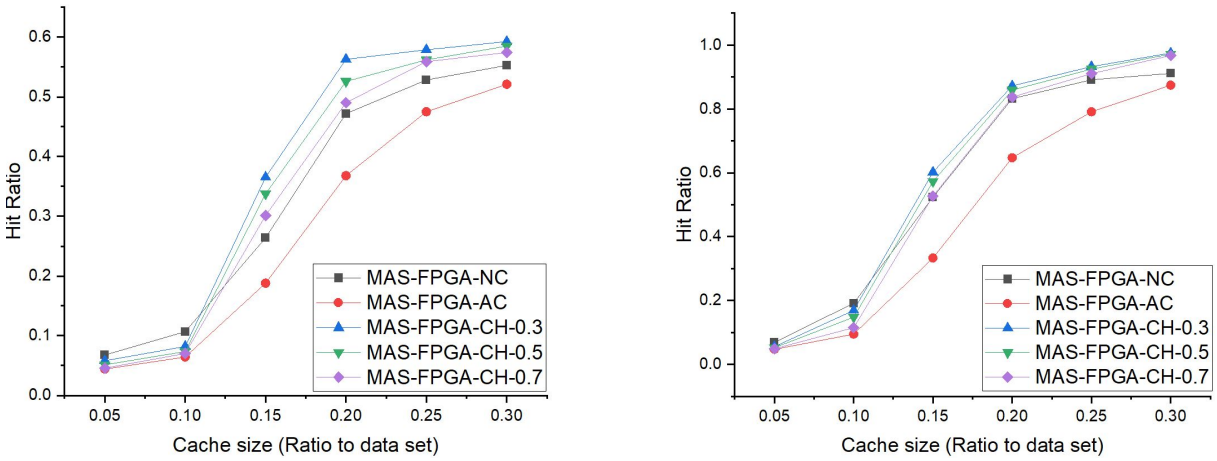
Figure 4.25 – Response times of Query Trimming accelerators with respect to different coalescing strategies.

In Figure 4.25a with workload of customized Q6(s), Query Trimming of MAS-FPGA with Coalescing Heuristic, named MAS-FPGA-CH-QT, is on average 6.4 times faster than Query Trimming of MAS-Server with Always Coalescing (i.e., MAS-Server-AC-QT). Meanwhile, if using Always Coalescing on FPGA, such a kind of acceleration is slightly higher (i.e., 5.3 times). We also found the similar decrease of such acceleration for Query Trimming with workload of customized Q5(s) (in Figure 4.25b), in particular, it drops from 9.6 to 8.2 times when changing from Always Coalescing to Coalescing Heuristic on FPGA, compared to the best strategy for server (i.e., MAS-Server-AC-QT). In other words, although the Always Coalescing gives the best result for Query Trimming on FPGA, Query Trimming with respect to Coalescing Heuristic is still remarkable. Nonetheless, with Never Coalescing, Query Trimming of FPGA (i.e., MAS-FPGA-NC-QT) brings the lowest acceleration compared to the MAS-Server-CH-QT (i.e., on average 2.3 times). Thus, with the capability of Query Trimming accelerators, the Coalescing Heuristic now becomes competitive with the previous most popular approach, Always Coalescing. In fact, Coalescing Heuristic strikes the balance in making decision of coalescing, its number of generated

segments is slightly higher than the best case (i.e., Always Coalescing). By this way, it can gain other benefits, such as hit ratio and cache space usage that are presented in next sections.

d) Hit ratios with coalescing heuristic

Besides the performance (in response time), in this section we evaluate the hit ratio as one of the other benefits of Coalescing Heuristic (in Figure 4.26). Since hit ratio is impacted by the threshold T of Coalescing Heuristic, we assign T as 0.3, 0.5 and 0.7 respectively before starting MASCARA. Moreover, we reuse the impact factors that are presented in Table 3.6 of Section 3.4. Interestingly, hit ratio of MASCARA is not affected by using either FPGA or server, thus, here we present only the results on FPGA. Note that changing type of query from select-project to inner join query is negligible to hit ratio. Thus, it is reasonable enough to present the results with workload of Q6(s).



(a) Skew $SK = 0.5$. (b) Skew $SK = 0.9$.

Figure 4.26 – Hit ratio of different coalescing strategies on MAS-FPGA.

As it can be seen in Figure 4.26, hit ratio of all testing prototypes, Coalescing Heuristic included, are low when cache size is small since it cannot store the whole of Hot Region HR. Another reason is that frequency of replacement is high due to lack of space to store new data regions. Thus, small cache size always leads to a poor performance in all aspects (more details in Section 3.4).

When $Size\_C$ is large enough, hit ratios of different strategies become more obvious. In particular, with $Size\_C = 15\%$ of data set and $SK = 0.5$ as in Figure 4.26a, hit ratios are on average 26.4%, 18.8% and 33.5% for MAS-FPGA-NC, MAS-FPGA-AC and

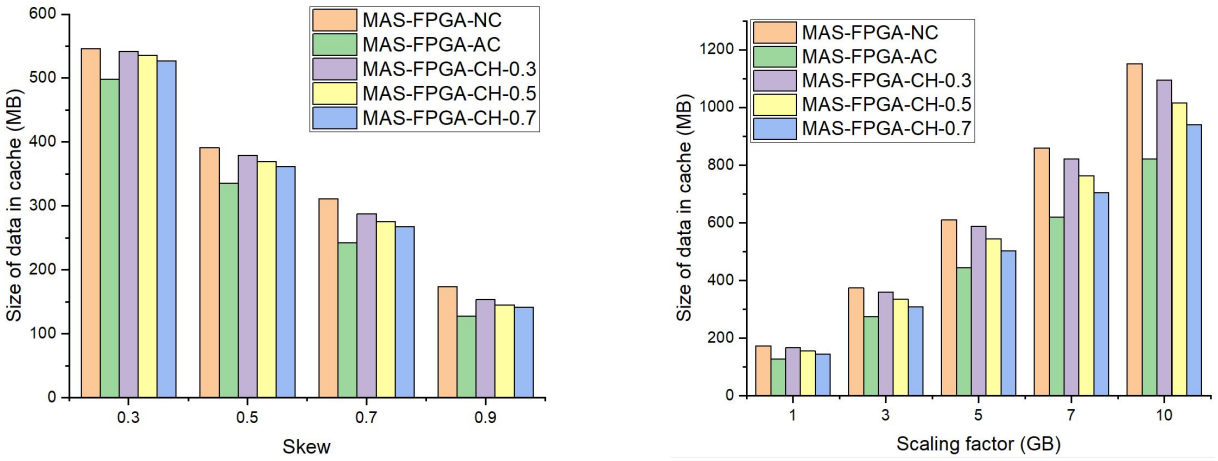
MAS-FPGA-CH, respectively. MAS-FPGA-AC brings the worst result in contrast to its performance in terms of response time. The reason is that a large merged portion from overlapped data regions DR could be replaced frequently in Always Coalescing. Thus, hit ratio can be seen as a trade-off of Always Coalescing. In contrast, MAS-FPGA-NC shows better results thanks to its higher data granularity in terms of a large number of tiny fractions of answer. Meanwhile, our solution, MAS-FPGA-CH gives the best results. In details, some of data regions with a high "profit" may not be coalesced in Coalescing Heuristic, thus, it maintains a balance number of fragments between Always Coalescing and Coalescing Heuristic. Moreover, by extending basic replacement value (i.e., LRU) in terms of "current profit" and "future profit" in Coalescing Heuristic, we strike spatial and temporal locality at the same time even if semantic locality of workload is low (i.e., $SK = 0.5$). Thus, hit ratio of Coalescing Heuristic is the highest among the coalescing strategies by having an appropriate number of segments that are most likely to contribute to the next queries.

We remind that T is used as a condition of filtering to decide for coalescing. In other words, increasing or decreasing T can affect the hit ratio which can be gained by Coalescing Heuristic. In details, low threshold (i.e., $T = 0.3$) means that the number of data region which could be merged together is decreased. In opposite, with $T = 0.7$, data regions have more opportunities to coalesce. Obviously, $T = 0.5$ can be seen as a balanced condition value of filtering. For example, with $Size\_C = 15\%$ and $SK = 0.9$ in Figure 4.26b, MAS-FPGA-CH-0.3, MAS-FPGA-CH-0.5 and MAS-FPGA-CH-0.7 present 60.2%, 57.3% and 52.7% of hit ratio, respectively. All of the strategies start to converge when cache size is large enough. Consequently, since there is only a slightly decline of acceleration when changing from Always Coalescing to Coalescing Heuristic, we agree it is worth to apply Coalescing Heuristic on MAS-FPGA to get higher hit ratios.

e) Cache space usage with coalescing heuristic

The last benefit of CH we want to discuss is the cache space usage. In details, coalescing or decomposing the data regions with respect to strategies can lead to different amount of spaces to be used in cache. Thus, we run the experiments over workload of Q6(s) to evaluate cache space usage for each coalescing strategies (in Figure 4.27). More precisely, the experiments in Figure 4.27a conducted with presented impact factors can be found in Table 3.6. Meanwhile, Figure 4.27b is presented based on Table 3.7. More importantly, capacity of SC can be seen as unlimited (but bounded by size of the card Alveo U2OO)

in all of the above experiments to simplify the measurement without considering cache replacement.



(a) SK is changed. (b) SF is changed.

Figure 4.27 – Content of cache with respect to applied coalescing strategies.

In Figure 4.27a with $SK = 0.3$ over $HR = 10\%$ as low hit ratio of cache, total used space in cache for all strategies is high. For example, total used space can be more than $500MB$ whatever the strategy is. Indeed, when miss happens frequently, MAS-FPGA receives more new results, thus it consumes more spaces at the end. In contrast, when hit ratio increases thanks to change of semantic locality (i.e., $SK = 0.9$ over $HR = 10\%$), cache consumes less space since only a few of the remainder queries with small answer could appear, in particular, total used space is less than $175MB$ for every strategy.

Even though space used in cache synchronously increases or decreases with respect to change of SK, there are still certain differences regarding applied coalescing strategy. More precisely, with $SK = 0.3$, spaces used in cache are on average $546MB$, $498MB$ and $534MB$ for MAS-FPGA-NC, MAS-FPGA-AC and MAS-FPGA-CH, respectively. Since cache size is unlimited, it is obvious that AC brings the best result in saving space of cache. Meanwhile, with NC, cache decomposes all of overlapped data regions with query answer to the new independent fragments. However, each of them has to store all of key attributes of the data set. Thus, this way causes an overhead of space in cache to store the duplicated key attributes for each of new data regions. Noticeable, when SK increases, space overhead of NC compared to AC gets worse due to more numbers of overlapped DR and larger size contributed to query answer. In particular, with $SK = 0.3$, MAS-FPGA-NC uses only 9.6% more spaces than MAS-FPGA-AC, but when $SK = 0.9$, this

gap increases to 36.3%.

Although CH does not bring the best results as AC in saving space of cache, it consumes less than NC. In particular, with $SK = 0.9$, MAS-FPGA-CH-0.3 uses 20.6% more spaces than MAS-FPGA-AC. Meanwhile, increasing T allows CH to coalesce more data regions. As a result, amount of duplicated keys can be reduced. Thus, $T = 0.7$ gives the best cache space usage meanwhile $T = 0.3$ is the worst case in using CH. Once again, choosing value of threshold T leads to the trade-off in hit ratio and space consumption of cache.

As it can be found in Figure 4.27b, when SF increases with fixed $SK = 0.9$, cache space usage increases for all strategies. This is the result of having many data regions DR with larger sizes that contribute to query answer when cache hit happens. More precisely, with $SF = 10GB$, NC now uses more than 39.2% spaces than AC. Meanwhile, the best case of heuristic, CH with $T = 0.7$ only uses more than 13.6% spaces than AC. Consequently, in all cases of SK, space overhead of MAS-FPGA-CH compared to MAS-FPGA-AC can be acceptable thanks to its high hit ratio that results to less execution of RQ in DMS.

f) Device utilization

In this dissertation, we analyze resource requirements on the device (i.e., Alveo U200) for the acceleration kernels of different query processing presented in MASCARA-FPGA (as shown in Table 4.1).

Type	Component	Number of units	LUTs	FFs	36KB BRAM
DB operators	Filter/Arithmetic	1 (64PCs and 32 RCs)	13366	9469	92
	Project	1	1454	927	-
	Sort Merge Tree-512 nodes	2	184672	31572	421
	Join	1	1449	762	-
Query Trimming	Attribute Matching	1	876	214	3
	Predicate Matching	16	21668	16621	114
	Semantic Extracting	1	1138	582	7
Cache manager	Query Process Controller (QPC)	1	1571	413	12
	Cache Filter: Heuristic	1	1263	629	10
Total resources			267457	61189	659
Percentage (%) of usage			25.48%	3.3%	37.29%

Table 4.1 – Hardware Resource Utilization of MASCARA-FPGA with one kernel.

This hardware resource consumption is given for the different MASCARA-FPGA's parts, such as Query Trimming and Probe Query Executing (as illustrated in Figure 4.1) on FPGA for one kernel. Looking into the components of Query Trimming in Table 4.1, the most expensive kernel is Predicate Matching which consists of heavy tasks as

computing units, such as Intersection, Difference and Implication. Currently, we deploy up to 16 instances for such procedure that works with 32 bit-width input streams of Range-Objects. Moreover, BRAMs are used a lot to store the results of computations and forward to next round (i.e., Semantic Extracting). In contrast, Attribute Matching has a small resource cost since its iteration is limited by the set of attributes which appear in the projection condition of the query. Indeed, the total number of (range-type) attributes in three relations *lineitem*, *orders* and *customer* are less than 15. Moreover, matching an attribute is a simple comparison over string type data. Similarly, Semantic Extracting also requires a small amount of resources with its straightforward implementation for if-else statements.

In MASCARA-FPGA, we control query processing and manage data coalescing in cache through two small components on FPGA. In particular, query plan of MASCARA-FPGA is static to simplify the executing. Thus, a simple QPC is reasonable for presented input queries without considering aggregate or complex join operations. Meanwhile, filter of coalescing heuristic maintains the "profit" value in terms of an arbitrary precision fixed-point data type 32 bit-width instead of floating type. Therefore, with the above elements, these components do not demand a large amount of resources.

Meanwhile, regarding to DB operators, Filter which includes several kinds of computations, requires a large number of LUTs. The reason is that it has to scan and evaluate streams of data in parallel by using a chain with 64 PCs and a network of 32 RCs as inputs. In opposite, Project consumes a few hardware resources (without considering scanning) due to its simple operation where column is selected by a masking mechanism. The most expensive accelerator is Sort-Merge in terms of sort-merge-tree. In fact, it can be configured with different depths, widths or ascending/descending orderings that results to changes on the report of resources used. In details, although nodes of tree (i.e., 2-to-1 MU) are tiny, an entire of tree which has multiple nodes can consume significant resources, especially BRAM utilization its proportional to the size of the tree. As it can be seen in Table 4.1, we deploy two sort-merge trees with 512 nodes. By this way, we ensure that all results from multi-view processing are sorted in parallel. In other words, we want to maximize the performance of sort-merge on FPGA. However, if we deploy only one tree, the acceleration of sort-merge is still acceptable compared to same computing on software implementation of server.

4.4 Conclusion

This chapter presents a cooperative model MASCARA-FPGA for multi-dimension range query processing to overcome the drawbacks (i.e., Query Trimming) of original MASCARA. Specifically, we design the Query Trimming accelerators to boost up the heavy computations, such as Intersection, Difference and Implication, between query and segments. Regarding to the design, computation tasks in terms of the iterations over the list of objects (i.e., CNF-Object, Range-Object) are presented from bottom-to-top parallelism. Additionally, we also implement DB operators, such as Filter and Project, that work with high-speed stream inputs to accelerate the execution of generated probe queries on FPGA. Regarding to the large scale size of processed data, off-chip memory of FPGA (i.e., DRAM) is used to cache and manage answers of queries in terms of *SC* management. In order to maintain the workflow between high level MASCARA and low level FPGA's accelerators, our prototype consists of other components, such as FPGA-Adapter and Query Process Controller (QPC). Furthermore, regarding the high throughput accelerators of MASCARA-FPGA, we reevaluate the benefits of our coalescing heuristic, especially the hit ratio and cache space usage. Last but not least, we prove that the bottleneck of Multi-view processing for inner join query on MASCARA is now overcomed by leveraging acceleration on FPGA.

In the experiments, we implement MASCARA-FPGA with single or multiple kernels (i.e., two) to evaluate how much acceleration can be gained compared to other solutions. Indeed, the results show that MASCARA-FPGA provides better performance (in response time) compared to Block-Cache and MASCARA-Server. In particular, MASCARA-FPGA with single kernel, can accelerate on average 2.9 and 6.9 times compared to Block-Cache and MASCARA-Server, respectively, using workload of Q6(s) from TPC-H. More precisely, Query Trimming on FPGA is 7.1 times faster than on CPU when complexity of query (e.g., Dimension of attributes) increases to seven. Meanwhile, the execution of (select-project) probe queries on FPGA is speed up to 8.5 times compared to CPU. According to Multi-view processing, sort-merge-join on FPGA is 7.7 times faster than software implementation on server, using workload of Q5(s) from TPC-H. In complement of semantic management, our CH applied on MASCARA-FPGA exhibits an acceleration up to 2.5 times compared to MASCARA-Server with Always Coalescing as the best case on server. Finally, CH also has the highest hit ratio (i.e., 33.5 with $SK = 0.5$ and $HR = 10\%$) and a balance space usage in cache compared to the two conventional approaches.

Chapter 5

Conclusion and Future Works

5.1 Summary of contributions

In this dissertation, we study the acceleration for range query processing in a new data management system (DMS) by deploying semantic caching (SC) framework in cooperation with an FPGA.

In particular, it seems relevant to consider a SC framework as cache management system (CMS) at the middleware layer of the DMS to ensure fine grained data re-usability and alleviate the problem of unnecessary query re-execution. Thus, we propose a ModulAr Semantic CAching fRAmework (MASCARA) where we divide and regroup functionalities of SC in terms of modules and stages. More precisely, this work is done by defining relevant templates, data structures, and interfaces. Thus, the main contribution of this architecture is its flexibility, scalability and adaptability to different environments, infrastructures and requirements. Moreover, MASCARA leverages the fast local storage of the compute layer meanwhile taking responsibility in communication with the (remote) storage layer of the DMS. We carry out extensive experiments based on data set generated from the TPC-H benchmark with various scenarios. The experimental results exhibit the performance of MASCARA in different aspects, such as response time, hit ratio and transferred data from storage. The best case shows that MASCARA is up to 3.9 times faster than the baseline (e.g., block cache). In contrast, we analyze a significant decline of response time in MASCARA when the query complexity increases in terms of dimensions, e.g., in the worst case MASCARA is 2.4 times slower than baseline. Therefore, we confirm that several heavy computing tasks of query rewriting in MASCARA will take advantages on FPGA

accelerators to enhance computing performances.

In order to bring more benefits for the SC framework, cache management of MASCARA is discussed in terms of coalescing strategy and replacement policy. In particular, we revisit the impact of conventional coalescing strategies, such as Always and Never Coalescing. By exploiting the strengths and mitigating the drawbacks from these conventional approaches, we propose a novel appropriate solution in cache management, named coalescing heuristic. In particular, it can decide when to coalesce data regions based on the recency of usage (temporal locality) and percentage of response contribution (spatial locality) that are presented through a new replacement function. Thus, it strikes a good balance between the two conventional approaches. Moreover, the heuristic can also increase hit ratio and reduce cache space usage of MASCARA. Importantly, we explain why the heuristic is not preferable to use with MASCARA based on CPU due to the complexity of the query rewriting. The experimental results show that Coalescing Heuristic is 2.3 times slower than Always Coalescing. We consider that this heuristic will become more noticeable when applying in MASCARA when it is accelerated by a specialized hardware (e.g., FPGA).

Processing select-project queries are seen as basic features of MASCARA. Meanwhile, join processing is more complex due to the participation of multiple of relations. Thus, we present an approach, named Multi-view processing, which brakes down an original (inner) join query into (select-project) sub-queries that belong to different joined relations or views. However, this approach can cause a significant execution time of Query Trimming in MASCARA due to the process of multiple generated sub-queries. The experimental results show that performance of MASCARA based on CPU can be reduced significantly. In particular, it runs 1.7 and 3.6 times slower than No-Cache and Block-Cache when the dimension of the query and the number of segments is high. Fortunately, we also explain that this issue can be overcome if MASCARA is accelerated with a specialized hardware (i.e., FPGA).

In addition to the need of accelerating Query Trimming, MASCARA is expected to leverage FPGA's capabilities, for example, low latency accelerators for the corresponding computing modules. Thus, we present a cooperative model between MASCARA and FPGA. To achieve this goal, we design the query rewriting of MASCARA and their tasks with respect to FPGA accelerators. We also develop the essential DB operators on FPGA, such as filter, project and sort-merge-join. We organize cache on off-chip memory (i.e., DRAM) of the FPGA which supports a reasonable capacity and high bandwidth connec-

tion to the accelerators. By coordinating all of them together, MASCARA-FPGA with single instance for every accelerator, is able to accelerate on average 6.9 times compared to MASCARA based on CPU. More precisely, Query Trimming on FPGA is 7.1 times faster than on CPU when the complexity of the query (e.g., Dimension of attributes) increases to seven. Meanwhile, the execution of (select-project) probe queries on FPGA is speed up to 8.5 times compared to CPU. According to Multi-view processing, sort-merge-join on FPGA is 7.7 times faster than the software implementation on the server. Since MASCARA-FPGA can handle the drawbacks of coalescing heuristic for semantic management and multi-view processing for (inner) join query, we revisit their benefits regarding to MASCARA-FPGA. On the one hand, MASCARA-FPGA with heuristic exhibits an acceleration up to 2.4 times compared to MASCARA-Server with Always Coalescing as the best case on the server. Additionally, heuristic has the highest hit ratio and a balance space usage in cache compared to the two conventional approaches. On the other hand, overcoming the presented issue of Multi-view processing, MASCARA-FPGA can maintain a high acceleration (e.g., on average 8.6 times for total response time) with inner join queries.

5.2 Future works

a) An extension of MASCARA-FPGA

MASCARA might be used to answer more complex queries in the future, such as outer join, ordering or top-n queries. To achieve this goal, our general idea to process such queries is to push all the basic operations that MASCARA can handle to the bottom of the query plan. Continuously, MASCARA could evaluate the remaining operators on the results of those selections. Multi-view processing is an instance for this approach where changing the order of filter and join operation does not affect to the final result. Certainly, other types (e.g., ordering) require more complex evaluations to prevent missing any semantic overlapping between query and segments. Otherwise, miss matching could happen and could reduce significantly the performance of MASCARA or waste cache space with duplicated data.

b) DB operators with respect to query plan

In this dissertation, MASCARA-FPGA is presented without implementing other database operators, such as hash join, group-by, and aggregation. Thus, it seems first interesting to extend the library of DB operators on FPGA to execute different kinds of (probe) queries. Second, such a kind of library may allow MASCARA to generate and select one of the execution plans which results to optimal performance of queries. This issue can be seen as an optimization in the (query) physical planning on FPGA. In particular, due to the resource limitation of FPGA kernels, query plan has to consider the possibility of using single or multiple FPGA kernels. To do that, a challenge is to propose an optimal query plan in a short time for FPGA specific cost model. To summarize, this model could consist of two components: a unit cost and an optimal query plan generation.

c) Satisfiability modulo theories SMT

We remind that satisfiability checking in Query Trimming can be seen as the NP-complete problem. Fortunately, a NP problem can be reduced with a polynomial time many-one reduction to the boolean satisfiability problem (SAT), in particular, the problem of determining whether a boolean formula is satisfiable [22]. Thus NP complete problem, which is reduced with SAT, can be resolved with SAT solver. Last recent years, the extension of SAT, named satisfiability modulo theories (SMT), from the boolean domain to various domains, such as the integer, real number or bit vector domains, has gained much attention. Thus, we consider that using SMT to check query equivalence in Query Trimming of MASCARA may be used. However, to use SMT solver in SQL expressions, we need to apply a transformation function from two-valued propositional or first-order logic into three-valued logical [18, 19]. By this way, we can model the semantics of widely-used SQL features, such as complex query predicates, arithmetic operations, and three-valued logic rather than simple range predicates. Additionally, FPGA also supports to convert SAT or SMT solvers into accelerators. In summary, a novel approach to implement Query Trimming which leverages on SMT solvers rather than an algebraic representation with respect to FPGA acceleration can be seen as a research challenge.

d) Optimization of cache management

In fact, reusing ratio of answers in cache depends on several factors, such as query workload characteristics, cache management, reuse algorithm, etc. Thus, in this disserta-

tion, we presented a coalescing heuristic with new replacement value function and make a comparison with other conventional approaches. Nonetheless, none of the presented coalescing strategies present a cost model to estimate and track completely the benefits of cache in terms of cached data regions, candidate for admission, reuse frequency of candidates, etc. It is worth to note that these above elements should be examined carefully with respect to changing the pattern of the workload or access statistics in the database. Thus, this work can be done to increase the usefulness of MASCARA-FPGA under any context of applications. However, proposing such kind of model could raise a detailed study which covers a lot of aspects, such as estimation of probability or approximate cost value functions (e.g., Knapsack problem, Dynamic Programming [11]). Obviously, it could result in a long latency both in coalescing strategy and replacement policy due to the complexity of computing and estimating the profit of data regions. Therefore, once again, it seems meaningful enough to leverage FPGA to accelerate partly or entirely this procedure.

e) Cache hierarchy

As a complement of the optimization problem, we also consider restructuring cache organization. In details, we can split the original cache into two children: fast cache (FC) and oracle cache (OC). On the one hand, OC can model and solve the computation issue of the cost model, which aims to find the nearly optimal set of data regions that should be retained. We assume that such kind of optimization problem is expensive and thus cannot be performed on a per-request basis. Thus, OC only recomputes its contents periodically, and mainly targets queries that have sufficient statistics in history. On the other hand, the FC cache serves as a fast "buffering" to temporarily store the answers for recent queries until OC collects sufficient statistics to make longer-term decisions. In other words, FC is optimized for an incoming query at that time, and can react immediately to workload changes. Collectively, the two caches can offer an efficient and reactive solution for MASCARA. Consequently, this work can be initialized with FC on server and OC on FPGA to accelerate the computations.

Appendices

The contributions of this thesis include the following publications:

1. Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau and Laurent d'Orazio. MASCARA (ModulAr Semantic CAching fRAmework) towards FPGA acceleration for IoT Security monitoring. International Workshop on Very Large Internet of Things (VLIoT 2020), Tokyo, Japan, 2020. Published in the Open Journal of Internet Of Things (OJIOT), 6 (1), Pages 14-23, 2020.
2. Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau and Laurent d'Orazio. MASCARA-FPGA Cooperation Model: Query Trimming through Accelerators. 33rd International Conference on Scientific and Statistical Database Management. Tampa, Florida, USA. Published in the Association for Computing Machinery, pp. 203–208, 2021.
3. Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau and Laurent d'Orazio. Cache Management in MASCARA-FPGA: From Coalescing Heuristic to Replacement Policy. 18th International Workshop on Data Management on New Hardware (DaMoN'22), Philadelphia, USA, 2022. Published in the Association for Computing Machinery, pp. 1-5, 2022.

List of Figures

List of Tables

Bibliography

[1] TPC Professional Affiliates. *TPC-H benchmark*. URL: http://www.tpc.org/tpch/.

[2] Amazon. *Amazon EC2*. URL: https://aws.amazon.com/ec2/?nc2=h_ql_prod_fs_ec2.

[3] Amazon. *Amazon ElastiCache*. URL: https://aws.amazon.com/elasticache/.

[4] Amazon. *Amazon S3 Cloud Storage*. URL: https://aws.amazon.com/s3/.

[5] Amazon. *Catapult High-level synthesis*. URL: https://s3.amazonaws.com/s3.mentor.com/public_documents/datasheet/hls-lp/catapult-high-level-synthesis.pdf.

[6] arcserve. *Scale Up vs. Scale Out Architecture: How Should You Scale Your Storage System?* URL: https://www.arcserve.com/blog/scale-vs.-scale-out-architecture-how-should-you-scale-your-storage-system.

[7] Michael Armbrust et al. « A View of Cloud Computing ». In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58.

[8] Michael Armbrust et al. « Spark SQL: Relational Data Processing in Spark ». In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394.

[9] Krste Asanovic et al. « A View of the Parallel Computing Landscape ». In: *Commun. ACM* 52 (2009), pp. 56–67.

[10] Edward Benson et al. « Sync Kit: A persistent client-side database caching toolkit for data intensive websites ». In: Jan. 2010, pp. 121–130.

[11] V. Boyer, D. El Baz, and M. Elkihel. « A dynamic programming method with lists for the knapsack sharing problem ». In: *Computers & Industrial Engineering* 61.2 (2011). Combinatorial Optimizatiion in Industrial Engineering, pp. 274–278.

[12] Sebastian Breß et al. « GPU-Accelerated Database Systems: Survey and Open Challenges ». In: *Trans. Large Scale Data Knowl. Centered Syst.* 15 (2014), pp. 1–35.

[13] Jared Casper and Kunle Olukotun. « Hardware Acceleration of Database Operations ». In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA, 2014, pp. 151–160.

[14] Ronnie Chaiken et al. « SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets ». In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1265–1276.

[15] Li Chen, Elke A. Rundensteiner, and Song Wang. « XCache: A Semantic Caching System for XML Queries ». In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: Association for Computing Machinery, 2002, p. 618.

[16] Boris Chidlovskii and Uwe Borghoff. « Semantic Caching of Web Queries. » In: *The VLDB Journal* 9 (Apr. 2000), pp. 2–17.

[17] Derek Chiou. « The microsoft catapult project ». In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 2017, pp. 124–124.

[18] Shumo Chu et al. « Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries ». In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1482–1495.

[19] Shumo Chu et al. « Demonstration of the Cosette Automated SQL Prover ». In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1591–1594.

[20] Sophie Cluet, Olga Kapitskaia, and Divesh Srivastava. « Using LDAP Directory Caches ». In: *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '99. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 273–284.

[21] Jason Cong et al. « Understanding Performance Differences of FPGAs and GPUs ». In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 93–96.

[22] Stephen A. Cook. « The Complexity of Theorem-Proving Procedures ». In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. URL: https://doi.org/10.1145/800157.805047.

[23] Louise H. Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Glasgow, GBR: Strathclyde Academic Media, 2014.

[24] European Union Agency for Cybersecurity et al. *Big data security : good practices and recommendations on the security of big data systems*. European Network and Information Security Agency, 2016.

[25] Laurent d'Orazio and Julien Lallet. « Semantic Caching Framework: An FPGA-Based Application for IoT Security Monitoring ». In: *Open J. Internet Things* 4.1 (2018), pp. 150–157.

[26] Laurent d'Orazio, Claudia Roncancio, and Cyril Labbé. « Adaptable cache service and application to grid caching ». In: *Concurrency and Computation: Practice and Experience* 22 (June 2010), pp. 1118–1137.

[27] Laurent d'Orazio and Mamadou Kaba Traoré. « Semantic Caching for Pervasive Grids ». In: *Proceedings of the 2009 International Database Engineering and Applications Symposium*. IDEAS '09. Cetraro - Calabria, Italy: Association for Computing Machinery, 2009, pp. 227–233.

[28] Laurent d'Orazio et al. « Building Adaptable Cache Services ». In: *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*. MGC '05. Grenoble, France: Association for Computing Machinery, 2005, pp. 1–6.

[29] Laurent d'Orazio et al. « Semantic caching in large scale querying systems ». In: *Revista Colombiana de Computación* 9 (June 2008).

[30] Shaul Dar et al. « Semantic Data Caching and Replacement ». In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 330–341.

[31] Databricks. *Scaling Genetic Data Analysis with Apache Spark*. URL: https://www.databricks.com/session/scaling-genetic-data-analysis-with-apache-spark.

[32] C. Dennl, Daniel Ziener, and J. Teich. « On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library ». In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines* 20 (2012), pp. 45–52.

[33] Christopher Dennl, Daniel Ziener, and Jürgen Teich. « Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration ». In: *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM*. Seattle, WA, USA, 2013, pp. 25–28.

[34] Prasad M. Deshpande et al. « Caching Multidimensional Queries Using Chunks ». In: *SIGMOD Rec.* 27.2 (June 1998), pp. 259–270.

[35] David J. DeWitt. « The Wisconsin Benchmark: Past, Present, and Future ». In: *The Benchmark Handbook*. 1991.

[36] Jian Fang et al. « In-memory database acceleration on FPGAs: a survey ». In: *The VLDB Journal* 29 (2019), pp. 33–59.

[37] Amir Gandomi and Murtaza Haider. « Beyond the hype: Big data concepts, methods, and analytics ». In: *International Journal of Information Management* 35 (Apr. 2015), pp. 137–144.

[38] Parke Godfrey and Jarek Gryz. « Answering Queries by Semantic Caches ». In: *Database and Expert Systems Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 485–498.

[39] Parke Godfrey and Jarek Gryz. « Semantic Query Caching for Heterogeneous Databases ». In: *Proc. KRDB Conf. Very Large Database*. Vol. 6. Sept. 1997, pp. 1–6.

[40] Jonathan Goldstein and Per-Åke Larson. « Optimizing Queries Using Materialized Views: A Practical, Scalable Solution ». In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: Association for Computing Machinery, 2001, pp. 331–342.

[41] Google. *Google Cloud Storage*. URL: `https://cloud.google.com/storage`.

[42] Sha Guo, Wei Sun, and M.A. Weiss. « On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems ». In: *IEEE Transactions on Knowledge and Data Engineering* 8.4 (1996), pp. 604–616.

[43] Sha Guo, Wei Sun, and Mark A. Weiss. « Solving Satisfiability and Implication Problems in Database Systems ». In: *ACM Trans. Database Syst.* 21.2 (June 1996), pp. 270–293.

[44] Alon Y. Halevy. « Answering Queries Using Views: A Survey ». In: *The VLDB Journal* 10.4 (Dec. 2001), pp. 270–294.

[45] Pat Helland. « Database Management System ». In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 714–719.

[46] Joseph M. Hellerstein. « Looking back at Postgres ». In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2019, pp. 205–224.

[47] Van Long Nguyen Huu et al. « MASCARA (ModulAr Semantic CAching fRAmework) towards FPGA Acceleration for IoT Security Monitoring ». In: *OJIOT* 6 (2020), pp. 14–23.

[48] ILNAS. *White Paper of Big Data*. URL: `https://portail-qualite.public.lu/dam-assets/fr/publications/normes-normalisation/information-sensibilisation/white-paper-big-data-1-2/wp-bigdata-v1-2.pdf`.

[49] MongoDB Inc. *MongoDB*. URL: `https://www.mongodb.com/`.

[50] Centrum Wiskunde & Informatica. *MonetDB*. URL: `https://www.monetdb.org/`.

[51] Alekh Jindal et al. « Selecting Subexpressions to Materialize at Datacenter Scale ». In: *Proc. VLDB Endow.* 11.7 (Mar. 2018), pp. 800–812.

[52] Björn Þór Jónsson et al. « Performance and Overhead of Semantic Cache Management ». In: 6.3 (Aug. 2006), pp. 302–331.

[53] Vinod Kathail. « Xilinx Vitis Unified Software Platform ». In: FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 173–174.

[54] A.M. Keller and J. Basu. « A predicate-based caching scheme for client-server database architectures ». In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. 1994, pp. 229–238.

[55] Dirk Koch, Daniel Ziener, and Frank Hannig. « FPGA Versus Software Programming: Why, When, and How? » In: June 2016, pp. 1–21.

[56] Dirk Koch, Daniel Ziener, and Frank" Hannig. « FPGA Versus Software Programming: Why, When, and How? » In: *FPGAs for Software Programmers*. Ed. by Dirk Koch, Frank Hannig, and Daniel Ziener. Cham: Springer International Publishing, 2016, pp. 1–21.

[57] Dongwon Lee and Wesley W. Chu. « Semantic Caching via Query Matching for Web Sources ». In: *Proceedings of the Eighth International Conference on Information and Knowledge Management*. CIKM '99. Kansas City, Missouri, USA: Association for Computing Machinery, 1999, pp. 77–85.

[58] Guoliang Li et al. « SCEND: An Efficient Semantic Cache to Adequately Explore Answerability of Views ». In: *Web Information Systems – WISE 2006*. Ed. by Karl Aberer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 460–473.

[59] Dalton Lunga et al. « Apache Spark Accelerated Deep Learning Inference for Large Scale Satellite Image Analytics ». In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 13 (2020), pp. 271–283.

[60] Zhenghua Lyu et al. « Greenplum: A Hybrid Database for Transactional and Analytical Workloads ». In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 2530–2542.

[61] Marouan Maghzaoui, Laurent d'Orazio, and Julien Lallet. « Toward FPGA-Based Semantic Caching for Accelerating Data Analysis with Spark and HDFS ». In: Aug. 2019, pp. 104–115.

[62] Rene Mueller, Jens Teubner, and Gustavo Alonso. « Data Processing on FPGAs ». In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 910–921.

[63] Rene Mueller, Jens Teubner, and Gustavo Alonso. « Glacier: A Query-to-Hardware Compiler ». In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. Indianapolis, Indiana, USA, 2010, pp. 1159–1162.

[64] René Müller, Jens Teubner, and Gustavo Alonso. « Sorting networks on FPGAs ». In: *The VLDB Journal* 21 (2011), pp. 1–23.

[65] Van Long Nguyen Huu et al. « Cache Management in MASCARA-FPGA: From Coalescing Heuristic to Replacement Policy ». In: *Data Management on New Hardware*. DaMoN'22. Philadelphia, PA, USA: Association for Computing Machinery, 2022.

[66] Van Long Nguyen Huu et al. « MASCARA-FPGA Cooperation Model: Query Trimming through Accelerators ». In: *33rd International Conference on Scientific and Statistical Database Management*. Tampa, Florida, USA: Association for Computing Machinery, 2021, pp. 203–208.

[67] Christopher Olston et al. « Pig Latin: A Not-so-Foreign Language for Data Processing ». In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 1099–1110.

[68] Oracle. *Java Native Interface JNI*. URL: `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html`.

[69] Oracle. *Javah tool kit*. URL: `https://docs.oracle.com/javase/9/tools/javah.htm#JSWOR687`.

[70] Oracle. *MySQL*. URL: `https://www.mysql.com/`.

[71] M. Owaida et al. « Centaur: A Framework for Hybrid CPU-FPGA Databases ». In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*. Napa, CA, USA, 2017, pp. 211–218.

[72] Andre Peters and Andreas Heuer. « BlueS: Usability of Semantic Caching Approaches in Pervasive Ad-Hoc Scenarios ». In: *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*. PETRA '11. Heraklion, Crete, Greece: Association for Computing Machinery, 2011.

[73] Gupta PK. *Accelerating Datacenter Workloads*. URL: `https://www.fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf`.

[74] Expassy: Swiss Bioinformatics resource portal. *UniProtKB/Swiss-Prot*. URL: `https://www.expasy.org/resources/uniprotkb-swiss-prot`.

[75] Tian Qin et al. « Performance Analysis of Nanoelectromechanical Relay-Based Field-Programmable Gate Arrays ». In: *IEEE Access* PP (Mar. 2018), pp. 1–1.

[76] Redis. *Redis enterprise cache services*. URL: `https://redis.com/solutions/use-cases/caching/`.

[77] Qun Ren and Margaret H. Dunham. « Using Semantic Caching to Manage Location Dependent Data in Mobile Computing ». In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. MobiCom '00. Boston, Massachusetts, USA: Association for Computing Machinery, 2000, pp. 210–221.

[78] Qun Ren, Margaret H. Dunham, and Vijay Kumar. « Semantic Caching and Query Processing ». In: 15.1 (Jan. 2003), pp. 192–210.

[79] Daniel J. Rosenkrantz and Harry B. Hunt III. « Processing Conjunctive Predicates and Queries ». In: *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*. VLDB '80. Montreal, Quebec, Canada: VLDB Endowment, 1980, pp. 64–72.

[80] Norvald H. Ryeng, Jon Olav Hauglid, and Kjetil Nørvåg. « Site-Autonomous Distributed Semantic Caching ». In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: Association for Computing Machinery, 2011, pp. 1015–1021.

[81] Behzad Salami et al. « AxleDB: A novel programmable query processing platform on FPGA ». In: *Microprocessors and Microsystems* 51 (2017), pp. 142–164.

[82] U.S. Securities and Exchange commission. *EDGAR Log File Data Sets*. URL: `https://www.sec.gov/about/data/edgar-log-file-data-sets.html`.

[83] Apache2 Server. *Log Files in Apache HTTP Server*. URL: `https://httpd.apache.org/docs/2.4/logs.html`.

[84] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. « Materialized View Selection for Multidimensional Datasets ». In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 488–499.

[85] Konstantin Shvachko et al. « The Hadoop Distributed File System ». In: MSST '10. USA: IEEE Computer Society, 2010, pp. 1–10.

[86] David Sidler et al. « DoppioDB: A Hardware Accelerated Database ». In: *Proceedings of the 2017 ACM International Conference on Management of Data*. Chicago, Illinois, USA, 2017, pp. 1659–1662.

[87] Deshanand P. Singh, Tomasz S. Czajkowski, and Andrew Ling. « Harnessing the Power of FPGAs Using Altera's OpenCL Compiler ». In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: Association for Computing Machinery, 2013, pp. 5–6.

[88] Ajitesh Srivastava et al. « A hybrid design for high performance large-scale sorting on FPGA ». In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–6.

[89] Divesh Srivastava et al. « Answering Queries with Aggregation Using Views ». In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 318–329.

[90] Michael Stonebraker et al. « On Rules, Procedure, Caching and Views in Data Base Systems ». In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. Atlantic City, New Jersey, USA: Association for Computing Machinery, 1990, pp. 281–290.

[91] Bharat Sukhwani et al. « Database Analytics: A Reconfigurable-Computing Approach ». In: *IEEE Micro* 34.1 (2014), pp. 19–29.

[92] M.R. Sumalatha et al. « Dynamic Rule Set Mapping Strategy for the Design of Effective Semantic Cache ». In: vol. 3. Mar. 2007, pp. 1952–1957.

[93] Bureau of Transport Statistics BTS. *Airline On-Time Statistics*. URL: `https://www.transtats.bts.gov/ontime/`.

[94] Andrei Vancea and Burkhard Stiller. « CoopSC: A Cooperative Database Caching Architecture ». In: *2010 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. 2010, pp. 223–228.

[95] Louis Woods, Zsolt István, and Gustavo Alonso. « Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading ». In: *Proc. VLDB Endow.* 7 (2014), pp. 963–974.

[96] Louis Woods, Jens Teubner, and Gustavo Alonso. « Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances ». In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, New York, USA, 2013, pp. 1073–1076.

[97] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. « Spatial Data Management in Apache Spark: The GeoSpark Perspective and Beyond ». In: *Geoinformatica* 23.1 (Jan. 2019), pp. 37–78.

[98] Qi Zhou et al. « Automated Verification of Query Equivalence Using Satisfiability modulo Theories ». In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1276–1288.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Contribution à l'accélération FPGA de cache sémantique pour le traitement des requêtes d'intervalles dans le domaine des masses de données

Mot clés : FPGA accélération, système de gestion de données, cache sémantique

Résumé :

Avec l'émergence de nouveaux systèmes de gestion de données pour le *big data et le cloud computing*, la mise en cache des données est devenue importante car elle permet de réduire l'exécution de requêtes inutiles. Dans ce contexte, le cache sémantique (SC) est une technique qui permet d'exploiter les ressources de la mémoire cache et les connaissances contenues dans les requêtes. Néanmoins, la réécriture de la requête avec un cache sémantique peut parfois induire un surcoût important en raison des calculs nécessaires. Dans cette thèse, nous cherchons à combiner l'infrastructure du cache, le cache sémantique et l'accélération de bases de données sur FPGA pour accélérer le traitement des requêtes d'intervalles dans le domaine des masses de données. Les contributions de cette thèse sont : 1) Nous présentons un système de gestion du cache dans la couche intermédiaire du système de gestion de données (MASCARA). 2) Nous proposons une heuristique de regroupement avec une nouvelle fonction de valeur de remplacement pour la gestion du cache dans MASCARA. 3) Nous mettons en œuvre un mécanisme, appelé traitement multi-vues, pour gérer la requête dites de jointure en cache sémantique. 4) Enfin, nous présentons un modèle coopératif, appelé MASCARA-FPGA, où le traitement des requêtes, en ce qui concerne la réécriture des requêtes et une partie de l'exécution des requêtes, est accéléré sur FPGA.

Title: Semantic caching framework towards FPGA acceleration for range query processing in domain of massive distributed data

Keywords: FPGA acceleration, data management systems, semantic caching

Abstract: With the emergence of new data management systems (DMS) in context of *big data and cloud computing*, caching data has become important since it can reduce unnescessary query execution. To address it, semantic caching (SC) is a candidate since it allows to exploit the resources in the cache and knowledge contained in the queries. Nevertheless, the complexity of query rewriting in SC, can induce a high overhead because of its excessive computations. Therefore, we aim to combine cache framework, SC and FPGA-based database acceleration together to accelerate range query processing in the domain of massive distributed data. In this dissertation, we present the contributions as follows: 1) We present ModulAr Semantic CAching fRAmework (MASCARA) in the middleware layer of DMS. 2) We propose a coalescing heuristic with a new replacement value function in terms of cache management in MASCARA. 3) We implement a mechanism, named Multi-view processing, to handle select-project-join query in SC. 4) We exhibit a cooperative model, called MASCARA-FPGA, where query processing is accelerated regarding query rewriting and part of query execution.