

A flexible and decentralised approach to query processing for geo-distributed data systems

Dimitrios Vasilas

▶ **To cite this version:**

Dimitrios Vasilas. A flexible and decentralised approach to query processing for geo-distributed data systems. Multiagent Systems [cs.MA]. Sorbonne Université, 2021. English. NNT : 2021SORUS132 . tel-03272208v2

HAL Id: tel-03272208
https://inria.hal.science/tel-03272208v2

Submitted on 21 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Sorbonne Université

École Doctorale Informatique, Télécommunications et Électronique

Scality

Laboratoire dInformatique de Paris 6

Inria

Équipe DELYS

A flexible and decentralised approach to query processing for geo-distributed data systems

Par Dimitrios Vasilas

Thèse de doctorat en informatique

Dirigée par Marc Shapiro et Bradley King

Présentée et soutenue publiquement le 19 février 2021

Devant un jury composé de :

M. Bernd Amann, Professeur, Sorbonne Université	Examinateur
Mme. Bettina Kemme, Associate Professor, McGill University	Examinateur
M. Bradley King, Co-founder & Field CTO, Scality	Encadrant
M. Sébastien Monnet, Professeur, Université Savoie Mont Blanc	Rapporteur
M. Themis Palpanas, Professeur, Université de Paris	Examinateur
M. Nuno Preguiça, Associate professor, Universidade Nova de Lisboa	Rapporteur
M. Masoud Saeida Ardekani, Software Engineer, Google	Examinateur
M. Marc Shapiro, Directeur de Recherche Émérite, Sorbonne Université & Inria	Directeur de thèse

Abstract

Query processing is an essential component of today's data serving systems. Query processing involves a variety of metrics that are in tension and create trade-offs. Because of these trade-offs, application developers need to tune query engines to the characteristics and needs of each application. Today's query engines often handle requests from users around the world, accessing data spread across geographically distributed sites. This thesis studies how to support efficient query processing in contexts in which users and data are distributed across multiple geographic locations.

We present an analysis of the design decision and trade-offs in geo-distributed query processing. In particular, we study how the placement of derived state used by the query engine to accelerate query processing (indexes, materialized views) and the communication patterns involved in query processing and state maintenance affect three metrics: query performance, query result freshness, and cross-site network resource consumption. We propose a query engine architecture that, as opposed to current state-of-the-art approaches, allows application developers to make derived state placement decisions in a case-by-case basis.

The enabling technique that this thesis presents is composition-based design: a query engine architecture can be constructed by composing building block components that encapsulate primitive query processing tasks into a directed acyclic graph that provides higher-order query processing capabilities. We introduce a query processing component abstraction, the Query Processing Unit (QPU), that defines a uniform interface and interaction semantics for query processing architecture building blocks. This uniform interface and interaction semantics allows us to expose design decisions about the query engines architecture and placement to application developers.

Finally, we present an implementation of the proposed approach, in the form of a framework for constructing and deployment application-specific query engines, called Proteus. Proteus consists of an extensible library of Query Processing Unit implementations, and mechanisms for facilitating the definition and deployment of QPU-based query engines.

The experimental evaluation supports the theoretical analysis of the trade-offs involved in query processing state placement, and suggests that Proteus can effectively occupy multiple different points in the design space of geo-distributed query processing.

Acknowledgements

One of my primary drivers for this work has been to contribute something that would be useful to other people. If I am even a bit successful in this, it is thanks to the collaboration, discussions and support from my advisors, colleagues, friends, and family. This section is my attempt in thanking each and every one of them.

It feels right to start by thanking my family. To my father, Kostas, thank you for passing on your life values to me, and by that making me who I am today. I hope I am living up to your expectations. To my mother, Georgia, and my sister, Christina, thank you for your care and support at my every step.

This work would not have been possible without the guidance, help and support of my advisors, Marc Shapiro and Brad King. I am thankful to them for always being patient, helpful, and insightful. Marc, thank you for all the time you dedicated discussing this work, refining the ideas, and challenging me to improve. Brad, thank you for teaching me something new about systems, their use, research, and even fluid dynamics in our every discussion. Thank you to both for teaching me how to do systems research and how to communicate this research so that it can useful to others.

I would like to thank the reviewers, Sébastien Monnet and Nuno Preguiça for dedicating the time to evaluate this thesis. I also thank the examiners, Bernd Amann, Bettina Kemme, Themis Palpanas and Masoud Saeida Ardekani for accepting to be part of my thesis committee. A special thanks to Sébastien and Themis for also accepting to be part of the thesis monitoring committee, and for offering their valuable feedback.

Research is not a solitary endeavour. Good research is only made possible through discussion and collaboration. I am fortunate to have been working among two great teams: Scality and the LIP6 laboratory.

I want to thank my colleagues at the Delys team. Alejandro Tomsic, Benoît Martin, Francis Laniel, Ilyas Toumlilt, Jonathan Sid-Otmane, Laurent Prosperi, Michael Rudek, Paolo Viotti, Saalik Hatia, Sara Hamouda, Sreeja Nair, Vincent Vallade and Vinh Tao Thanh, thank you for all the interesting discussions we have had over the years, and the time spent together.

Special thanks to Sara Hamouda for all her inputs, help and support with this work, and for being a great co-author.

Some special nods to Alejandro for the lunches at the Chinese restaurant, Ilyas for the Wednesday night runs, Jonathan whose the never-ending "I have something interesting to show you" were the beginning of fascinating endeavours, Saalik for his inexhaustible knowledge on technology, pop culture and more that always led to interesting discussions, and Sreeja for being someone I look up to as a researcher and a human being.

I would also like to thank the other Ph.D. students and interns at the LIP6 laboratory Alexandre, Antoine, Arnaud, Cédric, Damien, Daniel, Darius, Florent, Guillaume, Hakan, João Paulo, Lucas, Ludovic, Marjorie, Maxime, Redha and Sébastien, as well as the senior researchers, Jonathan, Julien, Luciana and Pierre, with whom I have shared my days at the laboratory.

I am grateful to the Scality family. I want to thank the Ironman team for welcoming me early on, and Vinh Tao and Vianney Rancurel for guiding me in my first steps in this thesis. I would like to thank the File Squad for welcoming me more recently, and for supporting and teaching me about working within an engineering team, and building a great product. A special thanks to Aline and Marina for supporting me through the challenges of moving to France. Thank you to Charles, Dogara, Duhamel, Florent, Greg, Imane, Jordi, Romain, Roxanne and Stéphanie for the workout sessions. Thank you to Alexandre, Anurag, Jordi and Partick for introducing me to bouldering. Also special thanks to Lam for all the discussions and the time we shared.

In addition, I would like to thank all the members of the LightKone European project for the exchanges and collaboration.

Georgios Goumas, Vangelis Koukis and Nectarios Koziris from the National Technical University of Athens in Greece inspired me and sparked my interest for understanding and building computing systems.

I also want to thank Stefanos Gerangelos for introducing me to the path of research, and guiding me through the first steps.

I am grateful to my close friends for accompanying me through this journey, and for supporting me, each in their own way, Aggeliki, Alexandros, Andreas, Argyro, Faidra, Giorgos K., Giorgos S., Giorgos V., Iliana, Kostis, Margarita, Maria and Thanos.

Finally, I owe a great deal to Kalliopi for her patience and understanding, for always being there to listen and for her ability to lift my spirits no matter the circumstances.

Contents

1 Introduction **5**
1.1 Contributions . . . 6
1.2 Thesis outline . . . 7

2 Preliminaries **8**
2.1 System Model . . . 8
2.1.1 Data storage tier . . . 9
2.1.1.1 System model . . . 9
2.1.1.2 Data model . . . 9
2.1.1.3 Data storage tier API . . . 10
2.1.2 Query processing tier . . . 11
2.1.3 Query language . . . 11
2.1.4 Derived state . . . 12
2.2 Query processing system performance evaluation . . . 13
2.2.1 Evaluating Efficiency . . . 13
2.2.1.1 Response time . . . 13
2.2.2 Evaluating Effectiveness . . . 13
2.2.2.1 Recall and precision . . . 14
2.2.2.2 Freshness . . . 14
2.2.3 Other aspects of query processing system design . . . 14
2.2.3.1 Availability . . . 14
2.2.3.2 Operational Cost . . . 15
2.3 Conclusion . . . 15

3 Background **16**
3.1 Query Processing in Relational Database Systems . . . 16
3.1.1 Materialized Views . . . 17
3.1.1.1 View maintenance . . . 18
3.1.1.2 Query Optimization and Materialized Views . . . 18
3.1.1.3 Materialized View Selection . . . 18
3.1.2 Distributed Query Processing . . . 18
3.1.3 Caching . . . 19
3.2 Query Processing in Non-Relational Database Systems . . . 19
3.2.1 Non-relational Database Data models . . . 19
3.2.2 Partitioning . . . 19
3.2.3 Replication . . . 20
3.2.4 Query Processing . . . 20
3.2.4.1 Secondary indexes . . . 21
3.2.4.2 Partitioning and Secondary Indexes . . . 22
3.2.4.3 Query Planning and Execution . . . 23
3.3 Conclusion . . . 23

4 The design space of geo-distributed query processing **24**
4.1 The use of derived state in query processing . . . 24
4.2 Design decisions and trade-offs in derive state based query processing systems . . . 25
4.2.1 Derived state maintenance schemes . . . 26
4.2.2 Derived state partitioning . . . 27
4.2.3 Derived state placement . . . 28
4.2.3.1 Geo-replication . . . 28
4.2.3.2 Geo-partitioning . . . 29
4.2.3.3 Multi-cloud and Query federation . . . 29

4.3 Conclusion . . . 30

5 A design pattern for flexible query processing architectures 31
5.1 Overview and design rationale . . . 32
5.2 The query processing unit: a building block for composable query processing architectures . . . 37
5.2.1 The Query Processing Unit component model . . . 37
5.2.1.1 Query, input stream, and domain interfaces. . . . 38
5.2.1.2 Configuration . . . 38
5.2.1.3 Local graph view . . . 38
5.2.1.4 Query Processing State. . . . 39
5.2.1.5 Initialization, query processing, and input stream processor methods. . . . 39
5.2.2 Query Processing Unit component model specification . . . 39
5.2.2.1 Query interface . . . 40
5.2.2.2 Query processing state . . . 42
5.2.2.3 Initialization method . . . 43
5.2.2.4 Query processing method . . . 43
5.2.2.5 Input stream processor method . . . 43
5.2.3 Stream semantics . . . 43
5.2.4 QPU classes . . . 44
5.2.4.1 QPU class case studies . . . 45
5.3 QPU-based query processing systems . . . 45
5.3.1 Query processing system architecture . . . 45
5.3.1.1 QPU-graph topology rules . . . 48
5.3.2 Query execution . . . 48
5.3.3 Query execution data structures . . . 50
5.3.3.1 Query parse tree . . . 50
5.3.3.2 Domain tree . . . 50
5.4 Discussion . . . 51
5.4.1 QPU-based query processing systems and ad-hoc queries . . . 51
5.4.2 Query processing unit consistency semantics . . . 52
5.5 Conclusion . . . 52

6 Case studies 53
6.1 Flexible secondary index partitioning . . . 53
6.1.1 Write path . . . 54
6.1.1.1 Graph topology and placement . . . 54
6.1.1.2 Index partition configuration and graph initialization . . . 54
6.1.2 Read path . . . 56
6.2 Federated secondary attribute search for multi-cloud object storage . . . 59
6.2.1 Predicate-based indexing . . . 61
6.3 Materialized view middleware . . . 61
6.3.1 Partial materialization . . . 64
6.3.2 Placing materialized views at the edge . . . 65

7 Proteus: Towards application-specific query processing systems 66
7.1 Query processing domain dissemination . . . 66
7.1.1 Domain interface . . . 66
7.1.2 Query processing domain discovery mechanism . . . 67
7.2 Query processing unit service . . . 68
7.2.1 QPU service architecture . . . 68
7.3 Architecture specification language . . . 70
7.4 Query processing system deployment . . . 71
7.5 Implementation . . . 71

8 Evaluation 73
8.1 Placing materialized views at the edge . . . 73
8.1.1 Experimental scenario . . . 73
8.1.2 Experimental Setup . . . 76
8.1.3 Query processing performance . . . 77
8.1.4 Freshness . . . 79
8.1.4.1 Freshness vs Throughput . . . 79
8.1.4.2 Freshness vs round-trip latency . . . 82
8.1.5 Data transfer between sites . . . 84
8.1.6 Conclusion . . . 84

8.2 Federated metadata search for multi-cloud object storage . . . 84
8.2.1 Experimental scenario . . . 84
8.2.2 Methodology . . . 85
8.2.3 Experimental Setup . . . 86
8.2.4 Query processing performance . . . 87
8.2.5 Freshness . . . 87
8.2.6 Data transfer between storage locations . . . 90
8.2.7 Conclusion . . . 90
8.3 Conclusions . . . 91

9 Related work **92**
9.1 Secondary indexing in distributed data stores . . . 92
9.1.1 Consistency between corpus and index . . . 93
9.2 Materialized views . . . 94
9.3 Result caching . . . 94
9.4 Geo-distributed query processing . . . 95
9.5 Modular & Composable architectures . . . 95
9.6 State and computation placement . . . 96
9.6.1 Computation placement . . . 96
9.6.2 State placement . . . 97
9.7 Distributed computation models . . . 97
9.7.1 MapReduce . . . 97
9.7.2 Dataflow engines . . . 98

10 Future Work and Discussion **99**
10.1 Future Work . . . 99
10.1.1 Placement policies . . . 99
10.1.2 Dynamic query engine architectures . . . 99
10.1.3 Consistent distributed queries . . . 100
10.2 Discussion . . . 100
10.2.1 Scope . . . 100
10.2.2 Fault Tolerance . . . 101
10.2.3 Data storage tier APIs . . . 101
10.2.4 Derived state in geo-replicated databases . . . 101

11 Conclusion **102**

Chapter 1

Introduction

Today's global-scale services handle large volumes of data and serve large volumes of requests from users distributed worldwide. They rely on large-scale data serving systems which distribute data across multiple geographically distant data centers in order to reduce response time and tolerate failures. Moreover, services increasingly spread their data between on-premise and remote cloud nodes, as well as between multiple cloud providers, in order to increase reliability and reduce costs.

An important aspect of the data systems that developers rely on for building applications is a powerful query language. Google, describing the evolution of Spanner from a key-value store to a relational database [22], has reported that Google developers found it difficult to build applications on a system lacking a strong schema system and a robust query language. Query processing is an essential component of data systems' technology, as its role is to bridge declarative query languages, in which queries specify the patterns and conditions that results must meet, and efficient execution.

In this thesis, we study the challenges of supporting efficient query processing in contexts in which users and data are distributed across multiple geographic locations. In particular, we examine the design decisions and trade-offs involved in the design of a query engine over geo-distributed data, which serves users distributed worldwide. We consider a query engine that implement operators for performing transformations over a corpus (selections, aggregations, joins) and, in addition, employs techniques for accelerating query processing. These techniques commonly involve the materialization of *derived state*. This includes maintaining copies of the data organized in forms that facilitate different access patterns (indexes), and pre-computing the results of expensive recurring operations (materialized views).

The design of such a query engine must meet multiple requirements. Applications expect query engines to serve requests in a timely fashion and to provide accurate results. Moreover, query processing must incur low overhead to other aspects of data systems' operation and to their operational cost.

Query processing in a geo-distributed context involves several challenges. In geo-distributed deployments, latencies between data centers can be orders of magnitude higher that those within a data center. Moreover, network resources between data centers are often limited and costly. Because of these factors, in geo-distributed query processing the requirements of low response time, accurate query results and low overhead to the system's performance and operational cost are incompatible and cannot be achieved all at once:

- User-perceived query response time is composed of two components: query network time, which accounts for the time spent to send a query from the user to the query engine over the network and the time spent to send search results back to the user, and query processing time. Techniques for reducing query processing time, like the above-mentioned, have been extensively studied in the context of database systems. However, in geo-distributed scenarios, query network time becomes a significant factor to query response time.
- At the same time, the derived state structures that query engines use to accelerate query processing must be kept up-to-date with changes to the corpus. This requires propagating and applying the effects of write operations to the derived state. Because of that, maintaining indexes and materialized views incurs overhead to the latency of write operations. An approach employed by query engines to address this issue is to perform this maintenance task asynchronously, outside of the critical path of write operations. This, however, entails that derived state is only eventually consistent, and can be temporarily stale relative to the corpus. Serving queries from stale indexes or materialized views may result in query results not being consistent with the state of the corpus.

Query network time can be reduced by replicating the query engine's derived state on every data center, thus ensuring that all queries can be served from the local data center without requiring communication with other data centers. However, maintaining a full replica of every index and materialized view on every data center incurs significant memory and storage overhead. Moreover, to remain up-to-date with

the corpus, a derived state replica must receive the effects of write operation from every data center. Assuming asynchronous maintenance, this means that derived state can become significantly stale, affecting the accuracy of query results. In addition, in write-dominated workloads, this results in significant data transfer between data centers for state maintenance.

At the other end of the design space, derived state can be partitioned into non-overlapping parts and each part assigned to a data center This approach improves derived state freshness. However, every query needs to be forwarded to every part of the derived state, resulting in significant overhead in query network time. Moreover, in query-dominated workloads this increases data transfer between data centers for query processing.

It becomes apparent that, in this setting, design decisions and about the distribution and placement of derive state and the communication patterns between the corpus, the users and the derived state, result in trade-offs between query response time, query result consistency, and the system's operational cost. Techniques such as caching of query results add additional points in this design space.

At the same time, modern applications have diverse query processing requirements. User-facing queries have strict response time requirements, as response times directly affect revenues. On the other hand, query engines that serve social media applications are required to ingest new content rapidly [32].

In addition to that, different design decisions about derived state distribution and placement are better suited to different workload characteristics. A design that requires only local communication between the corpus and derived state is better suited for write-dominated workloads in terms of data transfer costs. Conversely, a design that requires only local communication for query processing is more cost-effective for query-dominated workloads.

It therefore becomes evident that no single query engine design can be suitable for all needs. However, existing query engines are designed to be general-purpose systems; They aim at handling a variety of use cases and workloads. In the context of geo-distribution, general-purpose query engines are often inefficient for certain workloads and application requirements. Specialized solutions can be much more efficient, but adjusting derived state distribution and placement in current query engine architectures can be time and resource consuming.

In this thesis, we make the case for a *flexible* query engine architecture that can be adjusted to the requirements and characteristics of different applications in a case-by-case basis, with low engineering effort.

1.1 Contributions

Traditional monolithic query engine architectures do not provide the flexibility required to adjust the trade-offs between query response time, query result consistency, and operational cost according to the characteristics and requirements of different geo-distributed applications.

To address this challenge, we propose a new *modular* query engine architecture. Our key insight is that query processing can be decomposed into a set primitives, such as selecting from a set of data items the ones that satisfy a given condition, or constructing from a set of data items a secondary index on a given attribute. Each primitive can be encapsulated by an independent component that interoperates with other components through a set of interfaces. We show that, while different components encapsulate different query processing primitives, all components can expose a common set of interfaces with common semantics.

Guided by this insight, we introduce an architecture component abstraction, called Query Processing Unit (QPU). The QPU abstraction defines the properties, interfaces and semantics of a generic query engine component. In order to express different types of query processing primitives, the QPU abstraction combines the properties of a streaming operator and a microservice. Akin to a microservice, a QPU component maintains internal state, is configured independently, and exposes its functionality to other components through a service interface; Akin to a streaming operator, a QPU component operates by receiving one or more input streams from other components, performs a computation over these streams, and emits an output stream.

Different instantiations of the QPU abstraction (QPU classes) implement different query processing primitives. These include relational operators, such as selections, aggregations, and joins, derived state structures, such as indexes, materialized views and caches, and "routing" functionalities, such as load balancing, and managing index partitions.

Query Processing Units are aimed to be used as building blocks for constructing query engine architectures. A query engine is a directed acyclic graph (DAG), with QPUs as vertices; Graph edges indicate communication paths between QPUs. An edge from QPU_A to QPU_B indicates that QPU_A can invoke the interface of QPU_B, and that, as a result, it can receive an input stream from QPU_B. By specifying the QPU class and configuration of each graph vertex, as well as the graph topology, one can control how query engine's derived state is partitioned, and the communication patterns required for query processing and state maintenance. Because the operation of a QPU depends only on its interaction with other QPUs and not its placement, each QPU can be strategically placed across the system.

A QPU DAG runs a bidirectional data-flow computation. The corpus is situated at the leaf nodes of the graph, while queries enter the graph through root nodes. Updates to the corpus stream "upwards" [1] through the query engine graph, and incrementally update the QPUs internal data structures; Conversely, queries stream "downwards", are incrementally transformed in sub-queries which are processed in different parts of the graph. Partial results are then incrementally combined (flowing back upward) to produce the query response.

We realize this architecture model in the form of a query processing framework, called Proteus. Proteus provides a library of Query Processing Unit implementations, a service discovery mechanism that allows QPU graphs to self-organize with only local configuration to each QPU, and a configuration language for specifying query engine architectures.

Proteus aims at enabling developers to design and deploy query engines in a case-by-case basis, by enabling flexibility over the distribution and placement of the query engine's state and computations.

In summary, this work makes the following contributions:

- A comprehensive analysis of the design choices and corresponding trade-offs of geo-distributed query processing.
- A novel modular query engine architectural model, based on the Query Processing Unit abstraction, which enables flexible distribution and placement of query processing state and computations.
- Case studies that demonstrate how a flexible query engines architectures can be applied to a number of applications and provide improvements over state-of-the-art approaches.
- Proteus, a framework for constructing and deploying query engines using the proposed architectural model.
- An experimental evaluation that confirms the theoretic trade-off analysis, by demonstrating the benefits and drawbacks of different and placement schemes, and shows that Proteus can efficiently implement different points in the design space of query processing state placement.

Contributions of the thesis are presented in [141] and [142].

1.2 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 introduces the system and data model that we consider in this work, and discusses the requirements for an efficient and effective query engine design. Chapter 3 provides an overview of concepts related to the work presented in this thesis. It provides a broad overview of query processing in database systems, focusing on techniques for facilitating query processing by maintaining derived state (indexes, materialized views and caches). Chapter 4 presents an analysis of the design decisions and trade-offs involved in the design of query engines that derived state. Chapter 5 presents the specification of the Query Processing Unit abstraction, and the proposed query engine architecture model. Chapter 6 demonstrates the proposed architecture's expressiveness by applying it to a number of use cases and applications showing how it can provide improvements over the state-of-the-art. Chapter 7 describes the Proteus framework. Chapter 8 evaluates the proposed approach, and shows how flexible state and computation placement can allow applications to balance the trade-offs between query response time, result accuracy and operational cost. Chapter 9 overviews related work. Chapter 10 outlines directions for future work. Finally, Chapter 11 concludes the thesis.

[1] upwards and downwards are our representational convention

Chapter 2

Preliminaries

In this chapter we introduce the models and assumptions this work is based on. We define the system and data model that we consider in this work (§2.1), and discuss the requirements guiding the design of query processing systems (§2.2).

2.1 System Model

We consider a data serving system with a two-tiered architecture: a data storage tier and a query processing tier.

- The data storage tier is responsible for storing and providing access to data. We use the terms *storage system* or *data store* to refer to the system that implements this tier.
- The query processing tier is responsible for providing the functionality to identify and retrieve data using queries on secondary attributes (§2.1.2). We refer to the system that implements this tier using the terms *query engine* or *query (processing) system*.

Figure 2.1 shows a high level overview of the system model

- A user submits an *update operation* to the data storage tier. As a result, data flows from the user to the data storage tier, and then optionally to query processing tier. The query processing tier may use this data to update data structures such as secondary indexes and materialized views.
- A user submits a query to the query processing tier. As a result, control messages linked to the query execution flow through the query processing tier, and potentially for the query processing to the data storage tier (a query processed using cached data does not require control flow between tiers, while a query that requires reading from base tables does). Data, in the form of intermediate or final query results, flow from the storage tier to the query processing tier to the user.

Disaggregating query processing from the storage engine is an approach used by various systems and cloud services such as Amazon Athena [16], Aurora [17], and Google BigQuery [113]. In these systems, query processing is performed by a query engine independent from the storage system.

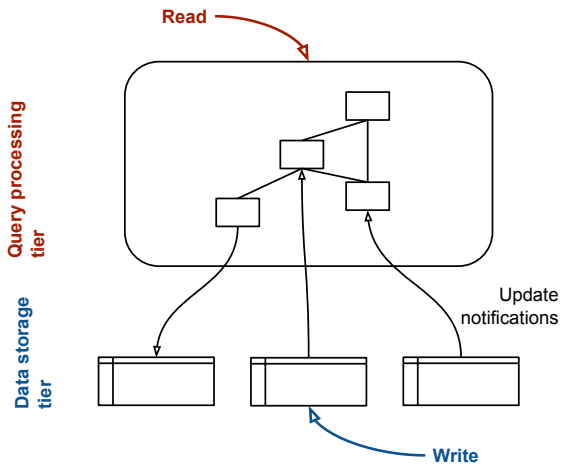


Figure 2.1: An overview of the system model.

This model provides several benefits:

- It enables application infrastructures to run a mix of different query workloads on the same data, without the need to move data to different systems in order to take advantage of their query processing capabilities.
- Storage and query processing resources can scale independently. The query engine can elastically adapt to match the query processing requirements of time-varying workloads, without over provisioning resources [143].
- It enables ad-hoc, one-time queries on already existing data without the need to migrate data. For example, performing log forensic queries for incident investigation.
- It enables cloud providers to implement fine-grained pricing for querying services.

In this work, we focus on the design of the query processing tier and consider the data storage tier as "imposed"; We consider the data storage tier's functionalities, guarantees, and data distribution schemes as input parameters in our design.

2.1.1 Data storage tier

The data storage tier is a broad abstraction which includes any system that can be used to store and retrieve data. This can include databases, file systems, cloud object storage systems.

As described above, in this work we view this tier as an external, underlying system that our design can build upon. However, any efficient query processing tier design needs to take into account the properties and characteristics of the data storage tier, most importantly its data distribution scheme. To address this, we adopt a narrower data storage tier definition. More specifically, we model a data storage tier as a federation of geo-replicated storage systems. In the following section, we present our data storage tier model in detail.

2.1.1.1 System model

The data storage tier is implemented by a storage system (a database, file system or cloud storage system) or a *federation* of multiple, potentially heterogeneous storage systems.

Each system is responsible for storing a large dataset D, continuously updated by a stream of small changes. We use the term *corpus* — traditionally used in the information retrieval literature to refer to a collection of documents — to refer to the entire collection of data stored by the data storage tier.

We model the storage systems that can be used to implement the storage tier as distributed, replicated partitioned systems [15]. In order to improve scalability, the corpus is divided into non-overlapping parts (usually call partitions or shards), so that each partition can be managed by a separate system node. Moreover, to improve availability, each partition is replicated. We consider both full replication, in which each replica is a complete copy of the corpus, and partial replication, in which a replica might be a copy of a subset of the corpus.

Finally, we consider federated systems that provide a unified namespace over multiple independent datasets stored in different storage systems.

We model the infrastructure on which the data storage tier runs as a collection of *sites*. A site is a group of nodes (servers, user devices) with the following characteristics:

- Network communication latency between nodes in different sites is significantly higher (typically an order of magnitude higher) [24] compared to communication latency within a site.
- Network bandwidth is high within a site and more limited and costly across sites. This is reflected in the pricing for cross-region data transfer in public cloud platforms. Using the AWS Pricing Calculator [1] we see that data transfer to different AWS data centers (regions) costs double the price of intra-DC data transfer (0.02 and 0.01 USD per GB respectively).

A site can correspond to a data center, a group of servers serving as an edge Point of Presence [6], or a group of user devices in close proximity (in the same room or building).

2.1.1.2 Data model

We define the corpus as a collection of *data items*, organized in *tables*. We use the term *data item* to refer to the unit of stored data: Depending on the semantics of the storage system that implements the data storage tier, a data item may correspond to a file, an object, or a database record. Tables may be organized in a hierarchical structure (file system directories), a flat namespace (object store buckets), or in a relational schema.

A data item is composed of a primary key, a set of attributes, and a value (or content). The primary key can be used to *efficiently* identify and retrieve data items, without requiring a scan. Attributes are key-value pairs. We do not assume a strict schema for attributes: the attributes of each data item are

independent of the attributes of others. Finally, a data item's value can be any sequence of bytes, such as an image, video, or PDF file.

Our main focus is on data item attributes. We consider data item attributes to be mutable. The data storage tier provides clients with operations for inserting and removing data items, modifying the attributes of a specified data item, and retrieving the attributes of a given data item.

This data model can express the data models of multiple different types of storage systems:

- Object stores, such as AWS S3:
  The data storage tier is implemented as an object store. In that case, a data item corresponds to an object and a tables to a bucket. The data item's value corresponds to the object's content, and attributes correspond to object tags [4].
- Wide-column stores, such as Amazon DynamoDB, Google's Bigtable or Apache Cassandra:
  In a data storage tier implemented by a wide-column store:
  - Data is organized in tables.
  - Data items correspond to table rows and attributes correspond to columns.
- Relational databases:
  In a data storage tier implemented by a relational database each table record corresponds to a data item, and each table column corresponds to an attribute. The relational schema can be represented by enforcing a corresponding schema for the attributes of data items in each table.
- Document-oriented databases:
  Document-oriented databases (or document stores) structure information in the form of documents (semi-structured data). Documents encode data in some standard format such as XML, YAML, JSON or BSON.

  The described data model is partially compatible with the document store data model: tables correspond to document collections, and data items correspond to documents. A document's identifier can be represented as a data item's primary key, while document attributes can be represented as attributes. However, the data model presented in this section cannot express complex attribute types such as lists and maps, or nested attributes which are used in the formats used in document stores.
- File systems:
  In a data storage tier implemented by a file system, data items correspond to files and tables correspond to file system directories. A data item's primary key corresponds to the corresponding file's path, the value to the file's content, and attributes correspond to extended file attributes.

This general data model, which is able to express different existing data models, satisfies the design goal mentioned above: Allowing the query processing tier to be independent from the underlying data storage tier, and be compatible with different storage tier implementations.

Timestamps
We assume multi-version storage. Each data item is associated with a timestamp. Modifying the value of a data item's attribute, creates a new version of the data item, which is associated with the timestamp assigned to the operation that resulted to it. Timestamps can be implemented using any data type that provides a partial or total order, such as Unix time or vector timestamps.

In summary, we model a table as a collection of tuples of the form:

$$(PrimaryKey,\ [Attribute],\ Timestamp)$$

where

$$Attribute = (AttributeName,\ AttributeValue)$$

Each tuple represents a data item version. $PrimaryKey$ is the data item's primary key, $[Attribute]$ is a set of attributes, each represented a name - value pair, and $Timestamp$ is the timestamp associated with that version.

2.1.1.3 Data storage tier API

One of our design decisions (Section 5.1) is to design the query processing tier as a middleware system, decoupled from the data storage tier. Moreover, we require the query processing tier to be able to interoperate with multiple different data storage tier implementations.

To achieve that, we model the interconnection between the data storage and query processing tier as a set of well-defined APIs. This allows the query processing tier to be compatible with any data storage tier implementation that exposes these APIs.

To be compatible with our design, a data storage tier needs to expose some the following APIs:

- An API for iterating over the corpus data (List). This API enables the query processing tier to access corpus data. It can be used to implement query processing tasks such as iterating over a table's data items and filtering those that match a given predicate, or performing a join over two tables.
- An API for subscribing to notification for changes to the corpus data (Subscribe). This API enables the query processing tier to receive a constant stream of notifications for corpus data changes, which can be used to incrementally maintain data structures such as indexes and materialized views.

For the rest of this document, we make the assumption that the data storage tier provides the List and Subscribe APIs as describe below.

List
The List API provides a mechanism for retrieving a set of versions of every object in a given table:

$$List(Table, [Timestamp_{low}, Timestamp_{high})) \rightarrow [ListResponse]$$

Given a table name ($Table$) and a time internal, expressed as range of timestamps ($[Timestamp_{low}, Timestamp_{high})$), $ListResponse$ contains all data item versions in $Table$ with $Timestamp_{low} \leq Timestamp < Timestamp_{high}$.

We do not assume specific semantics for $ListResponse$. For example it may be implemented as a single response containing an array of response entries, as a stream in which each entry is sent as a stream record, or an iterator in which calling a $Next()$ method returns the following response entry.

Finally, we do not assume any ordering in $ListResponse$.

Subscribe
The Subscribe API provides a mechanism for subscribing to notification for updates to data items in a given table:

$$Subscribe(Table, [Timestamp_{low}, Timestamp_{high})) \rightarrow [SubscribeResponse]$$

Given a table name ($Table$) and a time internal, $SubscribeResponse$ contains an entry corresponding to each update performed in a data item d in $Table$, with $Timestamp_{low} \leq Timestamp < Timestamp_{high}$. $SubscribeResponse$ entries are of two types. $Creation$ entries have the structure ($PrimaryKey$, [($AttributeName$, $AttributeValue$)], $Timestamp$). $Deletion$ entries have the structure ($PrimaryKey$, $Timestamp$). A creation entry represents an inserts operation that creates new a data item, while a deletion entry represents a delete operation. An operation that modifies the value of a data item's attribute is represented by a creation entry followed by a deletion entry.

$SubscribeResponse$ has streaming semantics: An invocation of $Subscribe$ returns a stream handler; The storage tier emits records at the stream corresponding updates performed to the corpus.

Various systems provide mechanisms that can be used to implement the $Subscribe$ API. Examples include triggers in traditional database management systems [125], and event notification mechanisms in cloud storage services [3].

2.1.2 Query processing tier

The main focus of this work is the design decisions and trade-offs involved in the design of the query processing tier. In this section we present an overview of the query processing tier's role and functionality. In the following chapters we review the known approaches and techniques for building a query processing tier (Chapter 3, examine the design decisions and associated trade-offs involved in the design (Chapter 4), and finally present our query processing tier design (Chapters 5 and 7).

The query processing tier is responsible for providing attribute-based data retrieval. It provides a $Query$ operation for retrieving data items using queries on their attributes.

2.1.3 Query language

We consider a query language that supports selection, projection, aggregation and join operators. We consider supporting an extensive query language to be out of the scope of this work. We argue that this query language is expressive enough to support a variety of applications, and simple enough to allow this work to focus on the architecture design of the query processing tier.

- The selection operator is defined as:

$$Selection(Table,\ AttributeName,\ Operator,\ Value)$$

where Operator is a binary operator in the set $\{<, \leq, =, \neq, >, \geq, >\}$, and $Value$ is a constant specifying a value. $Selection$ denotes all data items in $Table$ which have an attribute $AttributeName$, and for which $Operator$ holds between the value of attribute $AttributeName$ and the constant $Value$.

Moreover, queries can combine multiple $Selection$ operators with logical operators. For example, the query shown in Listing 2.1 specifies an *attribute predicate* using the conjunction of two $Selection$ operators.

- The projection operator is defined as :

$$Projection(AttributeName_0,\ AttributeName_1,\ ...)$$

$Projection$ operates in the set of attributes associated with a data item. The result of a projection operation over a data item, is the data item with its attribute set restricted to the intersection between the attributes specified by the projection, and the attributes associated with the data item.

- An aggregation operator is defined as

$$Aggregation(Table,\ Function,\ AggregationAttribute,\ GroupingAttribute)$$

where $Function$ is a function in the set {COUNT, SUM, AVERAGE, MAX, MIN}, and $OperandAttribute$ and $GroupingAttribute$ are attribute names. $Aggregation$ groups the data items in $Table$ in groupings based on their value of $GroupingAttribute$, and, for each group, applies $Function$ to the values of $AggregationAttribute$. The result of an aggregation function is a new table that contains a data item for each aggregation group. Each data item contains the result of the aggregation function as an attribute, and has the value of the grouping attribute as primary key. The query in Listing 2.2 groups the data items in Orders based on the value of the *customer_id* attribute, and applies the SUM function to the *amount* attribute of every data item in each group.

- The join operator is defined as

$$Join(BaseTable,\ JoinTable,\ JoinAttribute)$$

where $BaseTable$ and $JoinTable$ are two corpus tables, and $JoinAttribute$ the attribute on which to perform the join. $Join$ performs a left outer join. The join results contains all data items in $BaseTable$. For each data item in $BaseTable$, the operation examines the $JoinTable$ for *matching* data items. A matching data item is one that has the same value as the base data item for the $JoinAttribute$. If a matching data item existing for a base data item, then the join result for this data item contains the union of the attributes of the two.

We consider this type of join operation in our query model because it has been shown to be useful to application that model their data using semi-structured data [97].

```
SELECT order_id
FROM Orders
WHERE status != "shipped" AND order_date <= 10-10-2020
```

Listing 2.1: Query that retrieves order records based on their status and order date attributes.

```
SELECT customer_id, SUM(amount)
FROM Orders
GROUP BY customer_id
```

Listing 2.2: Query that computes the sum of the total amount for the orders of each customer.

2.1.4 Derived state

The principal functionality of the query processing tier is to provide the *Query* operation: it is responsible evaluating queries on secondary attributes over the corpus and returning the corresponding results. In this work, we focus on the techniques that the query processing tier employs to accelerate query processing: In particular, we consider the following techniques:

- **Caching:** In this work, we use the term cache to refer to an in-memory data structure that stores the results of earlier evaluated queries, so that future queries can be served faster. The query result cache that we consider in this work use a **write-around** writing policy: Write operations write to the data storage tier, and, for each write operation, the data storage tier sends an *update notification* to the query processing tier. An update notification is a message that encodes the changes in the corpus carried out by a write operation. Given an update notification, a query result cache performs one of the following, according to its *consistency policy*:
  - Performs no action.

  - Invalidates the cache entries affected by the changes in the corpus.
  - Updates the cache entries according to the changes in the corpus.

- **Secondary indexing:** When no index is used, every lookup on a secondary attribute requires a full table scan (iterating over every data item in a corpus table in order to determine which ones satisfy the given secondary attribute predicate). A secondary index is a data structure that accelerates this type of lookup: Creating a secondary index on an attribute, enables the system to retrieve the set of data items that have a specific value (or interval of values) from the index, without the need to perform a scan. The query processing tier updates secondary indexes using the mechanism of update notifications: For each write operation, the system updates secondary index entries so that they reflect the changes in the corpus.
- **Materialized views:** A materialized view is a data structure that *eagerly pre-computes* and stores the results of a specified query, at write time. The system can then serve that query by directly retrieving the results from the materialized view, which is less expensive than evaluating the query over the corpus. Similarly to the secondary index, for each write operation, the query processing tier receives an update notification, and updates the materialized view accordingly.

Given these definitions, a secondary index can be viewed as a specific case of a materialized view. In particular, a secondary index is equivalent to a materialized view for the query $Attribute = ?$.

In general, we refer to these data structures as *derived state*. Derived state, in the general case, is obtained by performing a computation over the corpus.

2.2 Query processing system performance evaluation

The aspects of a query processing system's performance can be categorized in two groups: *efficiency* and *effectiveness* [33]. We can measure efficiency with metrics such as response time, throughput, and scalability. Effectiveness is a measure of how well a query processing system achieves its intended purpose. It involves metrics such as precision (the fraction of useful information returned by the query) and recall (the fraction of data items in the corpus that satisfy a query returned by the query).

Finally, two other important factors in the design of query processing systems are availability and operational cost. Availability is important, due to the negative effects of downtime in user serving systems. It is especially relevant in the design of distributed query processing systems due to the multitude of faults that can impact the operation of a distributed system [78]. Moreover, the changes in system infrastructure brought by the cloud infrastructure model (fine-grained billing, independent scaling of different resources) allow more cost-effective system designs [143].

2.2.1 Evaluating Efficiency

In this work, we focus on applications that issue *interactive* queries, in which the most visible aspect of efficiency is the *response time* experienced by a client between issuing a query and receiving the corresponding response. Query processing systems that serve such applications need to be able to process large volumes of user requests, while keeping the response time of individual requests low. Because of that, an important efficiency metric is how response time scales as the system's throughput increases. The relation between response time and throughput, as well as between the offered load and the throughput achieved by the system, characterize the query processing system's *scalability*.

2.2.1.1 Response time

Response time, the delay between making a request and receiving the corresponding response, is among the most important metrics for the quality of a user-facing service.

A number of studies and experiments have studied the effects of response time to user experience. Results show that response time is among the factors with the most significant effect in users' subjective perception of the quality of a system. Users have been shown to perceive websites that load faster as more interesting [112]. On the other hand, long response times increase user frustration [39] and even compromise user's conceptions of the security of the system [28].

Industry reports have indicated that even small increases in user-perceived response times can result in drops in web traffic, and therefore sales. Experiments by the Google and Bing search engines have shown that longer page loading times have a significant impact on metrics such as time to click, repeated site usage, and queries per visit. A study from Akamai on the impact of travel site performance on consumers showed that more than half of the users will wait three seconds or less before abandoning the site [9].

2.2.2 Evaluating Effectiveness

Effectiveness is a measure of how well a query processing system achieves its intended purpose.

2.2.2.1 Recall and precision

In the field of information retrieval, which covers the problems associated with searching human-language data, the key notion linked with effectiveness is *relevance* [33]. In information retrieval, given a user's information need, represented by a search query, the *search engine* (the system responsible for query processing) computes a relevance *score* for each document (e-mail message, webpage, news article), and returns a ranked list of results.

Recall and *precision* are metrics often used to quantify the relevance of query results:

- Recall is the fraction of the relevant documents that are returned by the query. A recall value equal to 1 indicates that all relevant documents are returned by the query A recall value of less that 1 indicates that some relevant documents are not returned ("false negatives").
- Precision is the fraction of relevant documents among the documents contained in the query result. A precision value equal to 1 indicates that all documents returned by the query are relevant. A precision value of less than 1 indicates that some of the returned documents are not relevant ("false positives").

The difference between the information retrieval query model, and the query model that we consider in this work is in the *value space* of relevance. In information retrieval, relevance is a spectrum: Documents can be more or less relevant to a given query. Information retrieval systems assign a *score* to each document for a given query. In the scope of this work, we consider relevance as a binary metric: A data item is either relevant (satisfies the given query) or it is not. However, as described in Section 2.2.2.2, query results can, similarly to information retrieval, include non-relevant results (false positives), or not include relevant results (false negatives). Therefore, we argue that the recall and precision are meaningful metrics for evaluating the effectiveness of the query processing tier.

2.2.2.2 Freshness

Traditional database systems often keep derived state consistent with the corpus by updating both in a single transaction. For example, when executing an `UPDATE` statement, MariaDB updates a table's secondary indexes in the same transaction as the table rows [91]. However, in systems that implement asynchronous (lazy) derived state maintenance policies [111, 130, 135] derived state can become stale with respect to corpus.

Stale derived state may introduce false positives and false negatives to query results:

- **False positives**: A data item d that satisfied a query q has been deleted (or updated so that it does not satisfy q), but the corresponding derived state has not yet been updated to reflect this change. Serving q by reading from the stale derived state, includes d in response of q, introducing a false positive.
- **False negatives**: A data item d that satisfied a query q has been created (or updated so that it satisfies q), but the corresponding derived state has not yet been updated to reflect this change. Serving q by reading from the stale derived state, does not include d in response of q, introducing a false negative.

False positives and false negatives affecting the recall and precision of query processing.

We use the notion of *freshness* to refer to the measure of consistency between corpus and derived state due to asynchronous derived state updates.

A number of metrics for measuring data freshness have been proposed in the literature [30]:

- **Currency** measures the time between a change in the source data and that change being reflected in the derived state. In caching systems, the terms recency [31] and age [42] have been used to describe this metric.
- **Obsolescence** measures the number of updates to source data since derived state was last updated. Work on query systems has defined the *obsolescence cost* [60] of a query to represent the penalty of basing a query result on obsolescent materialized view. This cost is computed as a function of the number of insertions, updates, and deletions that cause deviation between the materialized view and the base table.
- **Freshness-rate** measures the percentage of derived state entries that are up-to-date with the source data. This metric has been used to quantify the freshness of web pages [81] and local databases copies [42].

2.2.3 Other aspects of query processing system design

2.2.3.1 Availability

The importance of availability becomes apparent when considering the negative effects of service downtime.

A study on user behavior in the Web [98] found that users abandon a non-working hyperlink after 5-8 seconds.

Operators of global services understand that "even the slightest outage has significant financial consequences and impacts customer trust" [49]. A 49-minute service outage in January 2013 cost Amazon an estimated $4 million or more in lost sales [10].

2.2.3.2 Operational Cost

Another important parameter that drives system design parameters is the system's operational cost.

Traditionally, the system is deployed on dedicated infrastructure, and the operational cost is the cost of owning and operating that infrastructure.

More recently, the Infrastructure-as-a-Service cloud computing models providing flexible, fine-grained provisioning of computing resources. These services provide fined-grained, "pay-as-you-go" pricing schemes, enabling more control over the system's operational cost.

A typical cloud pricing model [2] has distinct pricing for (1) computation and memory resources (vCPUs and memory), (2) persistent storage, and (3) data transfer.

2.3 Conclusion

This chapter presented the two-tiered system model that this work is based on, consisting of a data storage tier, and a query processing tier. It described the system's data model, and the query language provided by query processing tier. Finally, it presented the requirements and metrics that guide the query processing tier design.

Chapter 3

Background

In this chapter, we review concepts related to query processing in database systems. After presenting an overview of the "textbook" query processing techniques in relational databases in Section 3.1 we describe techniques aimed at reducing query processing time, including the use of materialized views (Section 3.1.1), and caching (Section 3.1.3). In Section 3.1.2, we describe state-of-the-art approaches of distributed query processing in commercial database systems. Then, in Section 3.2 we present an overview of research on query processing in non-relational database systems, focusing on concepts related to secondary indexing.

Query processing refers to the range of activities involved in extracting data from a database. These activities include the translation of queries from high-level languages into formats that can be processed by the database, query-optimizing transformations, and actual evaluation of queries.

Query processing has been studied extensively in the context of relational database systems. Relational databases provide sophisticated querying capabilities and require complex query processing techniques to connect declarative query languages to efficient query execution.

On the other hand, databases that belong in the class of non-relational database systems (also referred to as NoSQL) in general make design decisions aimed at high scalability (very large datasets or very high write throughput) and availability, and opt of more flexible and dynamic schemas than the relational schema [105]. A common technique used for supporting lookups on non-primary keys is secondary indexing [18, 90].

In this chapter, we review the query processing techniques used in both worlds, with the intend of identifying the primitives and approaches common in both classes of database systems.

3.1 Query Processing in Relational Database Systems

The database component responsible for query processing is the *query processor*. Given a declarative query (e.g. written in SQL), the role of the query processor is to validate it, optimize it into a procedural dataflow execution plan, and execute that plan.

Query processing consists of three main phases [72, 80]. First, the query processor parses the given query and generates a parse tree that represents the query's structure. Second, the query processor performs semantic analysis in order to transform the parse tree into a relational algebra expression tree. Finally, the query processor produces a query execution plan, which indicates the operations to be performed, the order in which they are to be evaluated, the algorithm chosen for each operation, and the way intermediate results are to be passed from one operation to another.
We describe in more detail the phases involved in query processing below.

Query Parsing. The goal of the query parsing phase is to translate a given query into an internal representation that can be processed by later phases, commonly a tree of relational operators [131]. While building the internal representation of the query, the query processor checks the query's syntax, verifies that the relation names that appear in the query are valid names of relations in the database, and verifies that the user is authorized to execute the query.

Query Rewrite. In the query rewrite phase, the query processor transforms the internal representation of the query in order to carry out optimizations that are beneficial regardless of the physical state of the system (the size of tables, presence of indices, locations of copies of tables etc.).

Typical transformations include:

- **Elimination of redundant predicates and simplification of expressions:** This includes the evaluation of constant arithmetic expressions, short-circuit evaluation of logical expressions via satisfiability tests, and using the transitivity of predicates to induce new predicates. Adding new

transitive predicates increases the ability of the following phase (query optimization) to construct query plans that filter data early in execution, and make use of index-based access methods.

- **View expansion, sub-query un-nesting:** For each view reference that appears in the query, the query processor retrieves the view definition and rewrites the query to replace that view with its definition. In addition, this phase flattens nested queries when possible.
- **Semantic optimization:** In many cases, integrity constrains defined by the schema can be used to simplify queries. An important such optimization is redundant join elimination (for example, a query that joins two tables but does not make use of the columns of one of the tables).

Query Optimization. In the query optimization phase, the optimizer (the query processor component responsible for query optimizations) transforms the internal query representation into an efficient plan for executing the query.

The optimizer is responsible for decisions such as which indices to use to execute a query, which algorithms to use to execute the join operations, and in which order to execute a query's operations. In a distributed system, the optimizer also decides about the placement of computations across the systems.

The foundational paper by Selinger et al. on System R [123] decomposes the problem of query optimization into three distinct sub-problems: cost estimation, relational equivalences that define a search space, and cost-based search. The optimizer assigns a cost estimate to the execution of each component of a query, measured in terms of I/O and CPU cost. To do so, the optimizer relies on pre-computed statistics about the contents of each relation, and heuristics for determining the cardinality of the query output. It then uses a dynamic programming algorithm to construct a query execution plan based on these cost estimates.

Query Execution All query processing algorithm implementations iterate over the members of their input sets. In [65], Graefe models these algorithms as algebra operators consuming zero or more inputs and producing one or more outputs.

A query execution engine — the query processor's component responsible for executing the query execution plan — consists of a collection of operators, and mechanisms to execute complex expressions using multiple operators. Query engines execute query plans by pipelined query operators; The output of one operator is streamed into the next operator without materializing the intermediate query results. An advantage of this model is that all iterators have the same interface; As a result, a consumer-producer relationship can exist between any two iterators; Operators can, because of that, be combined into arbitrarily complex query evaluation plans.

There are two approaches for implementing pipelining. traditionally, query execution engines have implemented *demand-driven pipelining*: Each time an operator needs an record it pulls the next record from its input operator, and wait input that operator produces the a record. That input operator might in turn require itself a record from its input operator, and so on. This approach has been popularized by its used in the Volcano system [64].

Conversely, in *data-driven pipelining*, records are pushed from source operators towards destination operators. This is commonly used in streaming systems.

Since query execution plans are algebra expressions, they can be expressed as trees; Tree's nodes are operators and edges represent consumer-producer relationships between operators. More generally, for queries with common sub-expressions, the query execution plan is a directed acyclic graph [65].

The concept of implementing a query execution engine as a directed acyclic graph of relational algebra operators, each executing a basic operation, forms the basis of our query engine architectural model. In Chapter 5, we describe how this concept can be generalized to include stateful operators that implement derived state structures, including indexes, materialized views and caches, as well as "meta-operators". We characterize as meta-operators (or routing) those operators that perform query processing control tasks, such as managing index partitions and load balancing, rather data manipulation tasks.

3.1.1 Materialized Views

An important element of the relational model is the *view*. A view is a "virtual relation" defined by a query that conceptually contains the result of that query. Views are not precomputed and stored; the database stores only the query defining the view. Views are computed *on-demand*; When a view is referred to by a query, the query engine expands it on-the-fly using its definition, and then processes the expanded query.

A materialized view is a view whose contents are *pre-computed* and stored by the database. In many cases reading the contents of a materialized view is much more efficient than computing the contents of the view by executing the query that defines the view. The use of materialized views is a common technique for improving query processing time.

3.1.1.1 View maintenance

An important aspect of materialized views is that when the underlying data referred in the view definition changes, the view must be kept up-to-date. A simplistic way of achieving this to re-compute the materialized view in response to every change to the corpus. A better option is to, given a change to the corpus, modify only the affected parts of the view. This approach is known as incremental view maintenance. Much research work has focused on incremental view maintenance in relational databases [87, 88, 153].

A design decision related to incremental view maintenance is *when* to perform the maintenance task: In *synchronous* view maintenance, view maintenance is performed as soon as an update occurs, as part of the updating transaction; In *asynchronous* or lazy view maintenance, updating the view is deferred to a later time [151]. Materialized views with deferred view maintenance may be somewhat out-of-date with the corpus.

3.1.1.2 Query Optimization and Materialized Views

Materialized views add further consideration to query optimization:

- Rewriting queries to use materialized views. The query processor may produce a more efficient query plan by rewriting the query to make use of an available materialized view.
- Replacing the use of a materialized view with its definition. In some cases, replacing the materialized view with its definition in a given query, rather than directly reading from the view's contents, may offer more optimization options. For example, consider a case in which a materialized view does not include indexes that can be used to speed up a certain query, but the underlying relations do. Using the views definition instead of its contents enables query execution to take advantage of those indexes.

3.1.1.3 Materialized View Selection

Materializing an appropriate set of views and processing queries using these views can significantly speed up query processing since the access to materialized views can be much faster than recomputing the views. In principle, materializing all queries that a system may receive can achieve the optimal query response time. However, maintaining a materialized view incurs a maintenance cost. In addition, query results may be too large to fit in the available storage space. There is therefore a need for selecting a set of views to materialize by taking into account query processing cost, view maintenance cost and storage space. The problem of choosing which views to materialize in order to achieve a desirable balance among these three parameters is known as the view selection problem [69, 92].

3.1.2 Distributed Query Processing

So far we covered query processing from the perspective of a single-node database, without considering data distribution. However, data is inherently distributed [22, 82] and therefore query processing needs to efficiently operate on distributed data. In addition, query processing computations need to be able to be distributed and run in parallel on multiple nodes to achieve better scalability.

In Ingres [56], relations can be distributed across a collection of "sites". Query processing is based on *decomposing* queries into sub-queries that can be processed on a single site. The database uses query decomposition heuristics based on two optimization criteria: minimizing response time and minimizing network traffic.

Spanner [22] is sharded, geo-replicated relational database system. Spanner uses a *distributed union* operator in the query tree to represent query distribution. Distributed union is used to a sub-query to each shard of a table, and concatenate the results. It provides a building block for more complex distributed operators such as distributed joins between independently sharded tables.

When a query tree is initially created, a distributed union operator is inserted immediately above every table. In the query optimization phase, where possible, query tree transformations may pull the distributed union operator up the tree in order to push the maximum amount of computation to the servers. In the query execution phase, distributed union routes a sub-query request addressed to a shard, to one of the nearest replicas of that shard in order to minimize latency.

CockroachDB employs a mechanism for distributed query processing computation [107] (for example join, aggregation, or sorting) on multiple nodes in order to improve performance. In CockroachDB, a query plan is a tree of operators, termed *aggregators*: each aggregator consumes an input stream of records and produces an output stream or records. The key idea is that an aggregator splits the input stream into *groups*: the computation for each group is independent of the computation for other groups; the output stream is the concatenation of computation result for all groups. Since results for each group are independent, different groups can be processed on different nodes.

3.1.3 Caching

A widely used technique for reducing query load to the query engine and improve query response time is to cache the results of common-case queries, in order to avoid re-evaluating the query when the corpus is unchanged.

A common approach is to deploy a *caching layer* between the database system and the application. In-memory key-value stores such as Redis [83] and memcached [7] are often used for this purpose. However, in this case the application logic is responsible for the caching logic, including writing query results to the caching store, and invalidating or replacing cache entries as the underlying data change. This makes this approach complex and error prone [74].

3.2 Query Processing in Non-Relational Database Systems

The querying capabilities of a non-relational database mainly follow from their distribution model and data model. Thus different non-relational databases have varying querying capabilities.

To further discuss query processing in non-relational databases, we first briefly introduce the data models and data distribution techniques used in these systems.

3.2.1 Non-relational Database Data models

Key-Value Stores. A key-value store's data model is a map/dictionary of key-value pairs. As the structure of values is usually opaque to the database system, this data model only supports get and put operations (requesting and writing value using a key). Key-value stores in generally favor scalability over a richer data model and more complex query capabilities: the simple key-value model makes partitioning and locating data efficient, thus enabling these systems to achieve low latency and high throughput.

Document Stores. A document store is a key-value store that restricts values to semi-structured formats such as XML, YAML, JSON or BSON [5]. This enables more sophisticated data access capabilities: apart from retrieving an entire document from its key, documents stores support predicate queries (retrieving the keys of all documents that match a given predicate), and joins.

Wide-column Stores. The data model of wide-column stores is often depicted as a relational table with many sparse columns. More accurately, this data model can be described as a distributed, multi-level, sorted map. The first-level keys identify rows (row keys) and the second-level keys identify columns (column keys).

3.2.2 Partitioning

Partitioning is a technique for dividing a logical database into smaller distinct parts, called partitions, and spreading across several nodes. Partitioning divides both the database's content (corpus) as well as its computations. Each partition effectively acts as a database of its own, although there may be operations that involve multiple partitions. Different database systems use different terms to refer to what we here call partition, including *shard* [55, 96] *region* [71], *tablet* [43] and *vnode* [84, 137].

The goal of partitioning is to spread data and load evenly across nodes. When implemented efficiently, it enables horizontal scaling: doubling the number of nodes in the system should make the system able to handle double the volume of data, and should double the system's read and write throughput.

Partitioning has implications for query processing. Database records are assigned to partitions based on a *partitioning key*. Because of that, performing a selection operation on a key other than the partitioning key, requires every partition to perform a full scan of its records, and retrieve the ones that satisfy the selection predicate. A technique employed to more efficiently identify the requested records is secondary indexing. We describe secondary indexing in detail in Section 3.2.4.1.

The partitioning techniques commonly used in non-relational databases are range and hash partitioning.

Range partitioning. Range partitioning assigns an interval of keys to each partition. These ranges of key are not necessarily evenly spaced, because data might be unevenly distributed. Partition boundaries might be chosen manually by an administrator, or automatically by the database management system.

Within each partition keys are kept in sorted order. This has the advantage that range queries on the partitioning key are efficient: it is easy to determine which partitions contain keys of a given range, and within each partition the key can be treated as an index.

The downside of this partitioning scheme is that certain access patterns can lead to hotspots. Therefore, systems that use range partitioning need mechanisms for detecting and resolving hotspots.

Range partitioning is used by Bigtable and its open source equivalent HBase [61], RethinkDB, and MongoDB before version 2.4.

Hash partitioning. An alternative approach that avoids the risks of skew and hotspots is to use a quasi-random hash function to determine the partition for a given key. Hash partitioning assigns each partition a range of hashes — rather than a range of keys — and every key whose hash falls within a partition's range is handled by that partition.

This partitioning scheme is efficient at distributing keys fairly among partitions. The downside of this approach is that does not allow for efficient range queries, as adjacent keys are scattered across multiple partitions.

Hash partitioning is used in Amazon's Dynamo, MongoDB since 2.4 [95], Riak, CouchBase, and Voldemort.

3.2.3 Replication

Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. Replication improves availability by allowing the systems to continue working even if some of its parts have failed, and increases read throughput by increasing the number of machines that can serve read queries.

There are multiple different replication strategies. These strategies can be categorized based on two design decisions [66]:

- How are updates regulated. Is there single "master" replica responsible for processing updates to a given data item, or can any replica with a given data item update its copy?
- Are updates propagated between replicas eagerly or lazily?

A common approach to the first design decision is called *leader-based* replication. One of the replicas is designated the *leader* (also termed *master* or *primary*). Every write is sent to the leader. The leader determines the order in which writes should be processed, and sends the corresponding data changes to the other replicas (termed *followers*, *slaves* or *secondaries*), Followers apply those changes in the same order. Reads can be performed from any replica.

This approach is used in MongoDB, RethinkDB, and Espresso.

Leader-based replication has one main downside: as there is only one leader (when replication is combined with partitioning there is one leader per partition), and all database writes must go through it, if the leader is unreachable writes cannot be performed.

An extension of leader-based replication is to allow more than one replica to accepts writes. In *multi-leader* replication there are multiple leaders, each processing writes and forwarding the corresponding data changes to all other replicas. Each leader acts also as a follower to the other leaders, accepting writes from them.

An alternative approach, termed *leaderless* replication, is to allow any replica directly accept writes from clients. Each write is sent (either by the client, or by a coordinator) to W replicas, and each read is sent to R replicas, where W and R are configuration parameters. In order to ensure that eventually all data is propagated to every replica, leaderless replication implementations often employ two mechanisms: (1) read repair, a way to detect and update stale values during reads, and (2) anti-entropy, having a background process that replicates missing data between replicas.

This approach was popularized by Amazon's Dynamo. Riak, Cassandra, and Voldemort are datastores with leader replication models inspired by Dynamo.

There are two approaches for the design decision "when the leader propagates data changes to followers". In *synchronous* (or *eager*) replication the leader propagates changes synchronously and waits for acknowledgements from followers before reporting success to the user. In *asynchronous* (or *lazy*) replication the leader propagates changes and does not wait for responses from followers.

The advantage of synchronous replication is that followers are guaranteed to have copies of the data that are up-to-date with the leader. Its disadvantage is that if followers do not respond (due to a crash, network fault or other reasons) writes cannot be processed.

Geo-replication (replication across geographically distributed data centers) can protect the system against data center failures and network problems, and improve read latency for clients distributed across multiple geographic locations. Synchronous geo-replication, as implemented in Google's Megastore [25] and Spanner [22], achieves strong consistency at the cost of high write latency. In asynchronous geo-replication, as used in Dynamo [49], PNUTS [44, 46], Walter [133], COPS [89], Cassandra [85], and Bigtable [40] the inter-data center network delays are hidden from clients, and the system remains available during partitions. The downside of asynchronous geo-replication is that the same data may be concurrently modified in different data centers creating conflicts that then need to be resolved.

3.2.4 Query Processing

Non-relational database systems in general support two types of queries:

Primary key lookups. In a primary key lookup, a data item is retrieved using its primary key. This is the main data access method in non-relational databases. It can be efficiently supported as it is compatible with both hash and range partitioning.

Predicate queries. A predicate query returns all data items from a database table that meet a condition specified over their attributes. In its simplest form, a predicate query can be performed as a filtered full-table scan.

3.2.4.1 Secondary indexes

For databases that use hash partitioning a full-table scan implies a scatter-gather operation where each shard performs a filtered scan, and results from all shard are merged.

A common technique used to support efficient predicate queries is the use of secondary indexes.

A secondary index is a structure that is derived from the primary data, and organizes data in a form that provides a way to efficiently access database records by means other than the primary key.

Typically, a secondary index consists of an entry for each existing value of the indexed attribute. Entries can be viewed as a key-value pairs, where the key is a value of the indexed attribute, and the value is a list of pointers to database records that contain this value (a *posting list*). In lower level index implementations, employed in centralized database systems or in the scope of a single partition, a pointer indicates the position of a record in the physical representation of the database. In higher level index implementations, typically used in distributed databases, the primary key of the record is used as pointer; This assumes the presence of a mechanism that can efficiently retrieve records via their primary keys. In this work, we consider this second pointer implementation.

Secondary indexing is an instance of a general system design pattern: having the same data represented in different formats to address different access patterns. Database tables are the primary copy of data. Derived copies of the data transform the primary copy differently in order to satisfy certain access patterns. Adding a secondary index does not affect the contents of the database; it only affects the performance of read and write operations. Writes go to the primary data and all of the other data copies are derived from it. The other copies only serve read requests.

Index data structures.
The following data structures are commonly used as secondary indexing structures:

B-Tree. The B-tree is the most widely used indexing structure. Its purpose is to keep key-value pairs sorted by key, which allows efficient key lookups and range queries. The B-tree breaks the indexed key-value pairs into fixed-size *pages* (traditionally 4 KB in size); reads and writes are performed in the granularity of a page. Pages can be identified using an address, which allows one page to refer to another, in disk instead of in memory. The B-tree uses these references to construct a tree of pages. Each page contains multiple keys and references to child pages. Each child is responsible for a continuous range of keys; keys between child page references indicate the boundaries of those ranges.

To update the value of an existing key in a B-tree, one must search for the leaf page that contains that key, change the value in that page, and write the page back to disk. Adding a new key consists of finding the page whose range contains the new key, and adding it to that page.

The B-tree algorithm ensures that the tree remains balanced: a B-tree with n keys always has a depth of $O(logn)$

Log-Structured Merge Tree. Like the B-tree, the log-structured merge (LSM) tree is a key-value structure that keeps keys sorted. An LSM-tree is composed of two or more tree-like component data structures. A smaller component (for example a red-black or AVL tree), sometimes called a *memtable*, resides entirely in memory. The rest of LSM tree's components are persisted on disk as Sorted String Table (*SSTables*). An SSTable is a sequence of key-value pairs, sorted by keys.

Write operations are performed on the memtable. When the memtable reaches some size threshold, the system writes it out to disk as an SSTable file. To serve a lookup, the LSM tree algorithm first tries to find the requested key in the memtable, then in the most recent on-disk segment, then in the next-older segment etc. A background process periodically merges SSTables by removing redundant and deleted keys and creating compacted SSTables.

LSM-trees are typically able to sustain higher write throughput that B-trees, partly because they sequentially write compact SSTable files to disk rather than having to potentially overwrite several pages for each write [34].

Originally the log-structured merge tree index structure was described by O'Neil et al. in [101]. The terms *memtable* and *SSTable* were introduced by Google's Bigtable paper [40]. LSM trees are used in data stores such as LevelDB [48] and RocksDB [27], and similar storage engines are used in Cassandra and HBase [26].

The B-tree and LSM-tree can be both used as primary or secondary index structures.

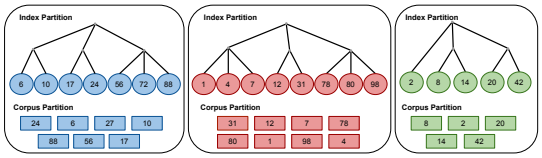


Figure 3.1: Index partitioning by document.

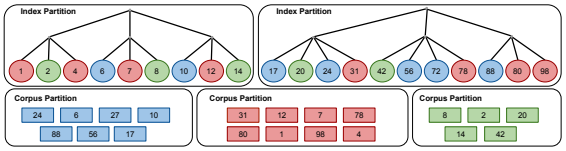


Figure 3.2: Index partitioning by term.

Figure 3.3: Two approaches for partitioning a secondary index, given that the corpus table is partitioned by its primary key. In the partitioning by document approach the index is partitioned so that each index entry of a data item is co-located with the data item. In the partitioning by term approach the index partitioned so that each index partition contains index entries in a particular range.

In this work, we focus on the aspects of employing secondary indexes on distributed data. We consider these aspects orthogonal to the index implementation; We abstract index implementation details by modeling a secondary index as a system component that provides the following APIs:

- An efficient range query operation $query(key_1, key_2) \rightarrow [value]$, where key_1 and key_2 are the boundaries of a range of keys. A point lookup is a special case in which $key_1 == key_2$.
- Operations for inserting, updating, and deleting keys.

We argue that this model can be used to represent any secondary index data structure. Our prototype implementation (Chapter 7) uses of-the-shelf state-of-the-art index data structure implementations.

3.2.4.2 Partitioning and Secondary Indexes

The partitioning schemes discussed in Section 3.2.2 rely on a key-value data model. Secondary indexes do not neatly map to these partitioning techniques: a secondary index usually does not uniquely identify a data item, but rather provides a way of searching for occurrences of a particular value.

There are two main approaches to partitioning a secondary index: document-based partitioning and term-based partitioning.

The terminology used in the rest of this section comes from the literature of full-text indexes (a particular kind of secondary index): a document is a self-contained piece of information, is composed of terms.

Partitioning Indexes by Document. In this approach, each partition is separate: each partition maintains its own secondary indexes, covering only documents in that partition.

In a document-partitioned index each database write (adding, removing, or updating a document) is handled only by the partition that contains the corresponding document. Reading from a document-partitioned index requires a scatter/gather approach: sending the query to all partitions and combining the returned results. This can make index lookups quite expensive. Even if index lookup requests are sent to partitions in parallel, response time depends from the latency of the slowest index partition.

This approach is commonly used in commercial systems, including MongoDB [127], Riak [18], Cassandra [90] Elasticsearch [138], Solr [132].

An index partitioned using this approach are commonly referred to as a *local index*.

Partitioning Secondary Indexes by Term. An alternative approach is to construct a logical "global" index that covers data in all partitions. A global index, however, needs to be partitioned itself, as storing it on one node would likely become a bottleneck.

To partition a global index, the indexed terms can be used as the partition key (thus the term *term-partitioned* index). Same as in corpus partitioning, the index partitioning scheme can use the terms themselves, which can be useful for range scans, or a the terms' hashes, which results to a more even load distribution.

The advantage of a term-partitioned index is that it can make reads more efficient: rather than requiring a scatter/gather over all partitions, a lookup for a given term only needs to make a request to the partition containing that term. The downside of this approach is that writes are more complicated and slower: a write to a single document may affect multiple partitions as the corresponding terms may correspond to multiple different partitions.

HBase [20] uses this approach: Secondary indexes are stored in regular HBase tables, using the indexed attribute as primary key. Term-partitioned indexes have also been used in the research systems such as SLIK [76] and Diff-Index [135].

An index partitioned using this approach are commonly referred to as a *global index*.

DynamoDB [146] and Apache Phoenix [21] support both local and global secondary indexes.

3.2.4.3 Query Planning and Execution

Most non-relational databases have simple query models that do not support complex operations such as aggregation and joins. However, some document-oriented databases like MongoDB [97], RethinkDB [8], and CouchDB [57] support join operations. Query planning in NoSQL databases mainly deals with the database's distribution model: a query execution plans consist of routing query requests to the appropriate data or index partitions.

3.3 Conclusion

This chapter presented an overview of concepts related to query processing. In particular, it presented query execution in relation database systems, focusing on the implementation of query execution engines. Furthermore, it described techniques used for improving query processing time, including materialized views, caching, and secondary indexing. The concepts presented in this chapter form the basis upon which the contributions of this thesis are built.

Chapter 4

The design space of geo-distributed query processing

Indexes, materialized views, and caches are crucial to query processing. Query engines often employ one or more of these technique to improve query processing performance. These techniques have in common that they proactively maintain *derived state* in order to achieve more efficient read access to data.

The use of derived state in query processing involves a number of design decisions, which are often in tension and create trade-offs. In this chapter, we analyze this design space: We present a unified framework for reasoning about how these design choices interact and how they affect the behavior of the query engine.

4.1 The use of derived state in query processing

Secondary indexing, caching, and the use of materialized views are all instances of a common technique: proactively applying a transformation to the corpus in order to accelerate anticipated queries.

At an abstract level, derived state can be described by a *write path* and a *read path* [78]. The write path is the process of creating the derived state, and keeping it up-to-date as changes to the corpus occur. The read path is the process of reading from the derived state in order to perform a query processing task. In other words, the write path is the pre-computation that takes place greedily, without knowing whether the results are going to be consumed; the read path is the computation that takes place lazily, when it is needed for query processing. Derived state is, therefore, an investment that pays off if the amortized overhead on the write path is less than the amortized gain on the read path.

We describe the read and write path of different derived state techniques using an example. Consider a database that stores images associated with user-defined tags. The database supports searching images by their tags. Consider the query:

```
SELECT * FROM photoAlbum
WHERE tags.predominantColor BETWEEN #0a6fb6 AND #52aca2
```

When no derived state is used, the read path evaluates this query by scanning all images in *photoAlbum*. If this is a frequent type of query, a database administrator can create a secondary index on the *predominantColor* tag to accelerate it. In that case, the read path performs an index lookup for the *predominantColor* predicate in the given query, while the write path updates the secondary index for each write operation.

Using an index moves an amount of work from the read to the write path: less work is required for serving the query (a fast index lookup rather than an expensive scan operation), at the expense of performing additional work each time a change in the corpus occurs.

Another option is to pre-compute query results. There exist two approaches for implementing this: caching and materialized views. In the case of caching, the read path attempts to serve the given query by reading from the cache. If the results are present in cache (cache hit), the system reads and returns them to the client. If not (cache miss), then the system evaluates the query and stores the results in the cache, so that it can serve future occurrences of the same query faster by retrieving the results from the cache. The write path depends on the consistency policy being used: We consider three policies (§2.1.4): performing no action on the write path, invalidating, or updating the cache.

In the case of materialized views, the system eagerly pre-computes query results in the write path. The read path retrieves the results from the view.

In summary, indexing, caching, and the use of materialized views shift the boundary between the read and write path. They allow query engines to reduce the work required on the read path, at the expense of performing more work on the write.

Tables 4.1, 4.2, and 4.3 summarize design space of derived state data structures, presenting the effect of each point of the space on the read path, write path, and the additional memory and storage resources required.

	Read path	
No derived state	Every query needs to be evaluated. Every lookup on a secondary attribute requires a full table scan.	
Caching	Fast path	Cache hit: Fast retrieval (no evaluation needed) of cached query results.
	Slow path	Cache miss: Fallback to other.
Secondary indexing	Fast path	Lookup on an indexed attribute: Fast lookup (no scan needed).
	Slow path	Lookup on a non-indexed attribute: Fallback to other.
Materialized views	Fast path	Fast retrieval (no evaluation needed) of materialized query results.
	Slow path	Fallback to other for non-materialized queries.

Table 4.1: Derived state design space: read path.

	Write path		
No derived state	No overhead for queries.		
Caching	No action on write policy	No overhead. Cache may become stale.	
	Invalidate/Update on write policy	Synchronous invalidation/update	Response time overhead on each write operation.
		Asynchronous invalidation/update	No overhead. Cache may temporarily become stale.
Secondary indexing	Synchronous maintenance	Response time overhead on each write operation that affects an indexed attribute.	
	Asynchronous maintenance	Index might temporarily become stale relative to the corpus.	
Materialized views	Same as secondary indexing.		

Table 4.2: Derived state design space: write path.

	Memory & storage resources
No derived state	No resource overhead.
Caching	Additional memory resources required.
Secondary indexing	Additional memory and storage resources required.
Materialized views	Additional memory and storage resources required.

Table 4.3: Derived state design space: memory and storage resources.

As discussed in Section 2.1.4, a secondary index, as defined in this work, can be viewed as a specific case of a materialized view. Because of that, throughout this chapter we use a secondary index as a running example The results also apply to materialized views.

4.2 Design decisions and trade-offs in derive state based query processing systems

Deciding whether using derived state is beneficial and choosing between the different derived state techniques involves a trade-off between read path work on the one side, versus write path work and the memory and storage resources overhead of maintaining derived state on the other. These trade-offs have been studied extensively [41, 139]. Database systems provide mechanisms for managing these trade-offs at runtime, by allowing the database administrator to select which indexes and materialized views to create, and by providing mechanisms to automate this choice.

However, in the context of geo-distribution, the use of derived state involves a number of additional design choices. Large-scale data serving systems, as discussed in Chapter 3, employ techniques such as partitioning and replication, both to the corpus and the derived state, in order to achieve low response time and improve scalability and fault tolerance. In the context of geo-distribution, there exist a number of additional considerations in the design of a query engine that employs derived state for accelerating query processing:

- Given a change to the corpus, when to update the derived state: synchronously in the same transaction, or asynchronously (§4.2.1)?
- How to distribute the derived state across the system's nodes (§4.2.2)?
- In system composed of multiple data centers, how to place derived state across data centers (§4.2.3)?

These design choices affect multiple aspects of the query processing system's behavior, including query performance, overhead to write operation response time, relevance of query results, and operational cost.

We can reason about how these design choices interact and how they affect the query processing system's behavior using the read and write path framework. To do so, we first describe a generic model for the read and write path, independent of a specific derived state data structure.

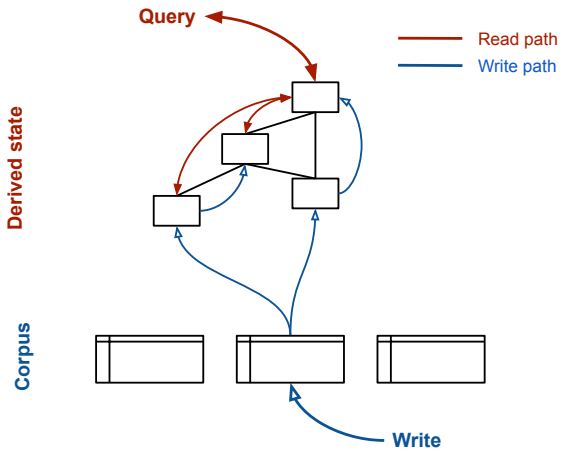


Figure 4.1: A conceptual depiction of the derived state's read and write path.

Figure 4.1 shows a conceptual depiction of a generic derived state structure's read and write path.

Read path. Given a query, the read path consists of sending a message (the query request) from the query source (the client or a query manager component) to the derived state, performing a query processing computation (for example an index lookup) at the derived state, and sending a message (the response) back to the query source. The query processing computation may itself involve additional communication, for example contacting other components of the query engine in order to request sub-query results.

Write path. Given a change to the corpus, the write path consists of sending a message (the update notification) from the corpus to the derived state, and the computation required to modify the derived state accordingly. Similarly, updating the derived state may involve additional communication between components of the derived state.

4.2.1 Derived state maintenance schemes

Given the following update to a data item d:

$$d.predominantColor := \#c78f83$$

updating a secondary index on the *predominantColor* attribute includes the following steps:

1. Insert d to the index entry that corresponds to $predominantColor = \#c78f83$.
2. Remove d from the index entry that corresponds to the previous value of $d.predominantColor$ (Depending on the protocol, the system might need to retrieve the previous value of $d.predominantColor$ from the data storage tier)

In practice, the derived state subscribes to the change log of the corpus. If the change log contains the information about the value of $d.predominantColor$ before the change, then step (2) is not required.

There are different design choices for *when* these steps are performed, relative to the critical path of the write operation. We call this design choice, the derived state's *maintenance scheme*.

In *synchronous maintenance*, all maintenance steps are performed in the critical path of the write operation, typically within the same transaction. This ensures that derived state is always up-to-date with the corpus, but increases the write operation's response time.

An alternative approach is to perform some of the maintenance steps *asynchronously*: deferring them for after the write operation has been acknowledged to the client. This reduces the overhead to write response time. However, it also allows derived state to be temporarily *stale*: For a given write operation, there is a time window in which a data item has been updated, but the update has not been reflected in

the derived state. As described in Section 2.2.2.2, serving queries from derived state that is stale relative to the corpus may result in query results with false positives and false negatives.

Amazon's DynamoDB maintains its global secondary indexes asynchronously. As a result, applications need to "anticipate and handle situations where a query on a secondary index returns results that are not up to date" [144].

We can reason about derived state maintenance schemes using the framework for derived state's read and write path. The system can perform tasks in the write path synchronously or asynchronously. Synchronous write path tasks incur overhead in write operation response time; Asynchronous ones relax the consistency between corpus and derived state. As a result, the choice of maintenance scheme involves a trade-off between write response time and query processing effectiveness (§2.2). The following asynchronous state maintenance schemes have been proposed in the literature:

Sync-insert

One asynchronous maintenance scheme consists of inserting new derived state entries synchronously, and removing the old state entries asynchronously in the background. The literature often refers to this scheme as *sync-insert*.

Sync-insert reduces the amount of work in the critical path of write operations, but it removes stale entries from the derived state asynchronously. Leaving stale entries in the derived state for a time window means that query results may contain false positives.

The sync-insert scheme is often complemented with a mechanism that, after reading from the derived state, removes false positives by validating the retrieved results against the corpus. This approach trades additional work in the read path, for improved query processing effectiveness.

Async-simple

In the *async-simple* asynchronous scheme, a write operation updates the corpus and returns immediately, deferring the derived state maintenance steps to later. In practice, async-simple is implemented using an asynchronous update queue: a write operation is acknowledged as soon as it is logged in the queue; a background process ingests the queue and updates the derived state.

This scheme incurs minimal overhead to write operations. However, because it performs all state maintenance tasks asynchronously, both false positives and negatives are possible. More specifically, considering an update that changes $d.attributeA$ from x to y:

- If none of the state maintenance steps have been performed, queries will return as if x is the value of $d.attributeA$ (false positive and false negative).
- If the entry for $d.attributeA = x$ has been removed from the derived state, but the new value has not been inserted, d will not appear in query results (false negative).
- If the entry for $d.attributeA = y$ has been inserted to the derived state, but the old value has not been removed, d will appear in query results for both values (false positive).

4.2.2 Derived state partitioning

Scaling derived state both in terms of size and read access requires partitioning it across multiple system nodes. In Chapter 3, we presented the two main index partitioning schemes. Briefly:

- In **partitioning by document**, index entries are co-located in the same node as the corresponding data items.
- In **partitioning by term**, the index is distributed independently from the corpus.

In the partitioning-by-document approach, an index lookup requires a scatter-gather operation: the system sends a lookup request to every index partition, and then combines the returned results. An advantage of this approach is that updating the index upon a write to the corpus does not require communication between nodes, since every data item is, by design, co-located with the index entries it may belong to.

In the partitioning-by-term approach, reading from the index requires contacting only index partitions that hold index entries relevant to the query. Updating a term-partitioned index, however, may require communication between nodes, as index entries may be located on a different node than the data item.

In summary, using the read and write path framework: Partitioning-by-document guarantees local-only communication on the write path, but requires a large volume of communication between nodes on the read path (a scatter-gather operation involving all partitions); Partitioning-by-term involves communication across nodes on the write path (a change to data item may need to be sent to an index partition on another node), but requires less communication on the read path in the general case (index partitions known do not include relevant index entries are not contacted).

Guided by this analysis, we can reason about the performance characteristics of the two approaches.

Partitioning-by-document is more suitable at a small scale (small number of nodes), while partitioning-by-term becomes more suitable as the number of nodes increases, and has been shown to provide better scalability [76].

In addition to the system's architecture scale, which approach is more suitable depends on a number of different workload characteristics. D'silva et al. [54] perform an extensive experimental comparison of the two approaches, implemented in HBase [70], and show how various workload characteristic affect the index partitioning scheme's performance.

The evaluation results show that:

- Partitioning-by-document is more suitable for: (1) smaller scale systems with a small number of nodes and low index lookup concurrency, (2) workloads with less selective queries that return large result sets, (3) skewed data distribution where a large number of data items have the same indexed value, or (4) write-intensive workloads.
- Partitioning-by-term is more suitable for: (1) larger scale systems with a greater number of nodes, (2) query-intensive workloads with a large query load, (3) workloads consisting of more selective queries with smaller result sets, or (4) data with normal distribution of the indexed value.

From these results, it is evident that neither of the two approaches is suitable for all needs. The choice of which approach to use should be guided by factors including the scale of the system, the properties of the corpus (distribution of indexed values over the data items), and the characteristics of the workload (query/write ratio, concurrency, query selectivity).

Clearly, the decision of which index partitioning approach to be used needs be taken in a case-by-case basis. We are not aware of any database system that provides this functionality. In existing database systems, the query processing system supports a single in index partitioning scheme (chosen by the database architect) [18, 76, 90, 135]; every index is partitioned using the same scheme. In Section 6.1 we present an approach for making the index partitioning scheme flexible, and exposing the choice of partitioning scheme as a configuration parameter to the application developer.

4.2.3 Derived state placement

So far in our analysis, we have considered a case in which corpus, derived state, and clients are placed on the same site, and therefore, that communication latency among them is not significant. However, in modern internet-scale services this is often not the case: Users are typically spread across multiple geographic locations; Database systems distribute or replicate data across geographically distributed data centers in order to minimize response time and to improve fault-tolerance.

Long-distance inter- data center network links exhibit latencies in the order of tens to hundreds of milliseconds [24]. This is an order of magnitude higher than latencies within a data center, and latencies experienced by users served from a data center close to their geographic location.

In this setting, design choices about 1) the placement of derived state, 2) the communication patterns between corpus and derived state, and 3) the communication patterns between derived state and clients, determine whether the read and write path computations require communication across data centers. Performing inter- data center communication in the read or the write path, incurs significant overheads. Inter- data center communication on the read path leads to increased query response time. Inter- data center communication on the write path either increases the delay between a write operation and the corresponding derived state update (in asynchronous maintenance), or increases the response time of write operations (in synchronous maintenance). Additionally, communication across data centers consumes cross-site network resources: In public cloud pricing models, cross-site data transfer incurs additional monetary cost. Therefore, in a geo-distributed setting, these design decisions play a major role in the query processing system's behavior.

Design choices related to inter- data center communication exacerbate the trade-offs presented in the previous Sections, but also pose additional trade-offs related to the replication of derived state. In this section, we discuss a number of scenarios that involve geo-distribution and the associated derived state placement approaches. We will use a secondary index as our running example; however, the analysis also applies to materialized views.

4.2.3.1 Geo-replication

Given a database that is replicated across multiple data centers, we consider the design choices for the architecture of a query engine that maintains a secondary index.

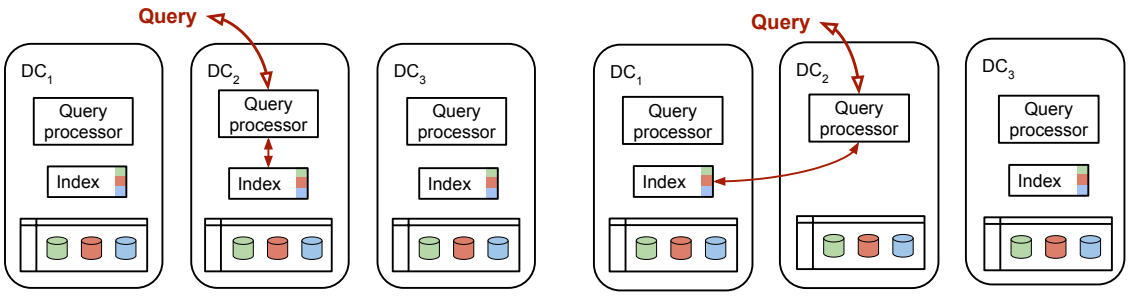


Figure 4.2: Index placement schemes over a geo-replicated corpus.

One approach is to maintain a replica of the index on each data center (Figure 4.2a). With this approach, both the read and write path involve only intra- data center communication: Queries can be served from the index at the closest date center, without requiring communication with remote data centers. Index maintenance is also local, since the entire corpus is present on each data center. The downside of this approach is the additional memory and storage resources required for maintaining index replicas.

An alternative approach is to place index replicas on a subset of the data centers. This reduces the storage overhead, but means that a query sent to a data center without an index either needs to be forwarded to another data center (Figure 4.2b), or be processed without using the index, both of which result in slower query processing. This approach is more suitable for cases in which some data centers receive more queries than others. Moreover, this approach can be supplemented with the use of caches in data centers without index replicas.

4.2.3.2 Geo-partitioning

A different geo-distribution strategy is geo-partitioning. In geo-partitioning, data is partitioned across data centers, and partitions are placed on data centers according to their access patterns. As a result, read and write access to data that belongs to a local partition is fast. CockroachDB for example, allows developers to partition data across data centers with row granularity [147].

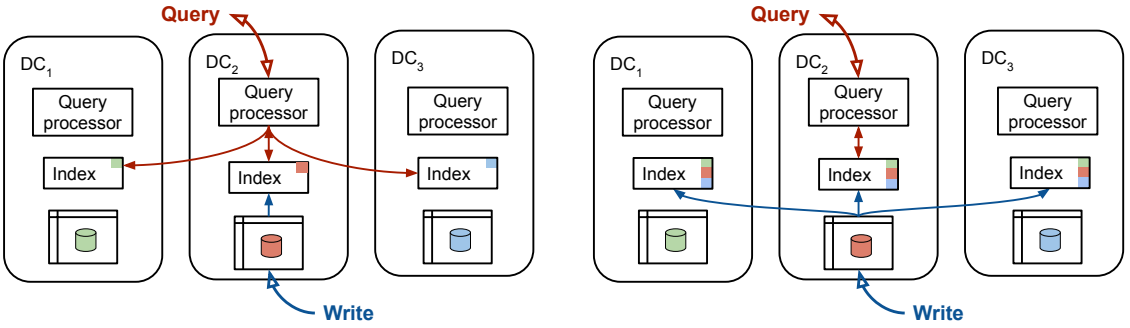


Figure 4.3: Index placement schemes over a geo-partitioned corpus.

One approach for maintaining a secondary index over geo-partitioned data is to use the partitioning-by-document scheme: each data center maintains a secondary index covering the local data items (Figure 4.3a). This approach has the same implications as document-based partitioning: the write path requires only intra- data center communication, since the index is co-located with the corresponding data items; the read path needs to contact the index partition on every data center, and gather the results.

A different approach is to maintain a global secondary index on each data center (Figure 4.3b). In that case, the index lookups (read path) can always be served locally. However, each change to the corpus needs to be propagated to all data centers (write path). The downside of this approach is that it significantly increases the memory and storage cost required for maintaining the index replicas.

4.2.3.3 Multi-cloud and Query federation

More recently, the concept of multi-cloud data storage has emerged. Organizations spread their data across private data centers and public cloud services in order to reduce costs and ensure fault-tolerance.

Moreover, they distribute data across multiple cloud providers in order to avoid dependence on a single cloud provider, take advantage of diverse storage and computing services, and improve reliability.

The advent of data distributed across multiple independent storage systems has created the need for unified access and federated search across platforms. The problem of federated query processing can be viewed as a generalization of the geo-partitioning case: the corpus at each platform can be seen as a partition of the logical global corpus. For simplicity, we assume that a multi-cloud platform consists of multiple, independent instances of a common database system. Under there assumptions, the same design decisions and trade-offs as in the case of geo-partitioning apply to query federation across multiple clouds. The same observations can be generalized to heterogeneous systems composed of different database system. However, federated query processing includes additional challenges linked to the heterogeneous data models of these systems.

From this analysis, it becomes clear that derived state placement and its communication patterns with the corpus and clients play a major role in the performance of geo-distributed query processing.

4.3 Conclusion

- The use of derived state to speed-up query processing moves query processing work from the read path to the write path, while also incurring memory and storage overhead. Moreover, different derived state schemes (indexes, materialized views, caching) entail different amount of work on the read and write path.
- The choice of derived state maintenance scheme controls the impact of work on the write path: Synchronous maintenance keeps derived state up-to-date with the corpus, but incurs overhead to the latency of write operations. On the other hand, asynchronous maintenance schemes move some of the work on the write path in the background. This reduces the overhead to write operations, but also means that derived state is not always up-to-date with the base data. Because serving queries from derived state that is stale relative to the corpus may introduce false positives and false negatives, design decisions on derived state maintenance pose a trade-off between the overhead to write operations and the relevance of query results.
- The derived state partitioning scheme determines the communication patterns between corpus and derived state on the write path, and between derived state and client on the read path. More specifically, partitioning-by-document ensures that no cross-node communication is needed on the write path, but requires a scatter/gather operation across all state partitions on the read path. Partitioning-by-term requires cross-node communication on the write path but reduces the number of state partitions that need to participate on the read path. Therefore, the choice of partitioning scheme poses a trade-off in the type (intra-node or cross-node) and volume of communication on the read versus the write path.
- When corpus and clients are distributed across different geographic locations, derived state placement affects communication latency (intra versus inter data center communication) between derived state and corpus and between derived state and clients. This creates a number of trade-offs, that result from the basic trade-off of requiring intra data center communication on the read versus the write path. These trade-offs include:
  - Low latency on the read path versus increased latency on the write path in the case of synchronous maintenance, or decreased freshness in the case of asynchronous maintenance.
  - Low latency on the read path versus increased resource utilization for maintaining derived state replicas, and increased volume of write path communication.
  - Low data transfer cost on the read path versus on the write path.

Chapter 5

A design pattern for flexible query processing architectures

We have so far established that query processing involves requirements that are in tension and create trade-offs. As a result, there can be no general-purpose solution to query processing. Rather, achieving efficient query processing requires the ability to *tune* the system to the requirements and characteristics of specific use cases. Existing query processing systems provide some tunning capabilities, for example index and materialized view selection.

However, as discussed in Section 4.2.3, in modern, internet-scale systems in which clients and data are geo-distributed, design decisions about *placement* of derived state (indexes, caches, materialized views) involve additional trade-offs and magnifies the effects of existing trade-offs. Existing query processing systems do not provide mechanisms for controlling the placement of derived state.

The design of a geo-distributed query processing system presents several unique design issues:

Tunability.
Achieving efficient geo-distributed query processing requires the ability to adjust the query processing system to the workload characteristics, data and access distribution patterns, and requirements of specific use cases. An efficient geo-distributed query processing system should provide tunning mechanisms along the following dimensions:

- *Flexible placement of derived state.* In geo-distributed environments, communication latency between sites is an order of magnitude higher than communication latency within a site. The use of secondary indexes and materialized views in this context, involves a trade-off between the requirements of minimizing the inter-site communication in the read path, and in the write path. We have have analyzed the implication of this trade-off in Section 4.2.3. Therefore, an efficient query processing system requires fine-grained control over the placement of derived state across the system.
- *Flexible placement of query processing computations.* For the same reasons, achieving efficient geo-distributed query processing requires the ability to control the placement of query processing computations in order to minimize the amount of inter-site communication. For example, the system can accelerate the execution of a join operation between two sets of data items located in different sites by performing the join in the site in which the larger set is located.
- *State maintenance scheme.* Keeping derived state up-to-date with the corpus involves a trade-off between the overhead incurred to write operations and the consistency between derived state and corpus data (Section 4.2.1). Updating the derived state synchronously ensures that it is strongly consistent with the corpus, but introduces additional work to the write path — potentially involving inter-site communication. Conversely, performing this task asynchronously reduces the impact to write operations, but allows derived state to be stale relative to the corpus. Which maintenance scheme is better suited to a specific use case depends on its workload characteristics (query or write-heavy), and consistency requirements.
- *Index and materialized view selection.* As we described in Section 4.1, the choice of which secondary indexes and materialized views to materialize involves a trade-off between work on the read path on the one side, and one the other side work on the write path as well as the resource required for maintaining derived state. Therefore, the query processing system needs to enable configurable index and materialized view selection.
- *Caching.* As we described in Section 4.1, caching, similarly to materialized views, can be viewed as a point in the design space defined by the read and write work framework. Therefore, index and view selection mechanisms should also include decisions about caching.

Independence from corpus distribution schemes.
Geo-distributed applications feature a variety of diverse data distribution schemes. Examples include full and partial data replication across data centers, geo-partitioning, and federated systems. An effective geo-distributed query processing system should not be limited to a specific data distribution scheme.

We argue that traditional monolithic query processing system architectures cannot provide the flexibility required to address these requirements — notably enabling flexible derived state placement. In this work we take a different approach. We propose a framework for constructing query processing systems in a case-by-case basis. The key idea is a modular, composable query processing architecture: application developers can construct and deploy query processing systems by assembling composable building blocks that encapsulate primitive query processing tasks. In this chapter, we focus on the *mechanisms* required for enabling a modular and composable query processing system architecture.

Parts of the work presented in this chapter have been published in the papers "A Modular Design for Geo-Distributed Querying: Work in Progress Report" [141] and "Towards application-specific query processing systems" [142].

5.1 Overview and design rationale

Our first design decision is to decouple the query processing architecture from the storage architecture. We design the query processing system as a *middleware system* that communicates with the storage tier through well-defined interfaces. This provides the query processing system with the flexibility required to support diverse data distribution schemes. Moreover, it allows the query processing system to be compatible with different storage systems, which is required in order to support federated query processing.

The second decision is to not focus on the design a *specific* query processing system, but rather a *framework* for constructing query processing systems in a case-by-case basis. The proposed framework provides a set of composable building blocks that encapsulate simple query processing tasks such selections, aggregations, and joins, as well as derived state structures, including secondary indexes, materialized views. These building blocks can be composed into modular query processing architectures. Moreover, we design the building blocks as independent components with well-defined boundaries. Because of that, their placement across the system is orthogonal to their functionality. This is key insight for achieving flexible state and computation placement.

Our design is based on a set of **observations** about the functionality and semantics of query processing systems. We use these observations as a basis for breaking down the functionalities of a query processing system into a set of *primitives*. We unify the semantics of these primitives under a component that can be used as the building block of a composable, modular query processing architecture.

In order to describe these observations, we introduce the example of a news aggregator application. In that application, users post and vote on news stories. The application stores individual votes for stories in a votes table. It uses a materialized view to compute the vote count of each story and join it with the rest of the story's records, as shown in Listing 5.1.

```
TABLE stories (id int, title text, url text)
TABLE votes (story_id int)

VIEW storiesWithVoteCount
  SELECT stories.id, stories.title, stories.url, v.vote_count
  FROM stories
  JOIN (
    SELECT story_id, COUNT(*) as vote_count
    FROM votes
    GROUP BY story_id
  ) v
  ON stories.id = v.story_id
  WHERE stories.id = ?
```

Listing 5.1: Definition of a materialized view that computes the vote count of each story and joins it with the rest of the story's attributes.

Moreover, this materialized view is partitioned with the partition by document approach, using the $story_{id}$ as partitioning key.

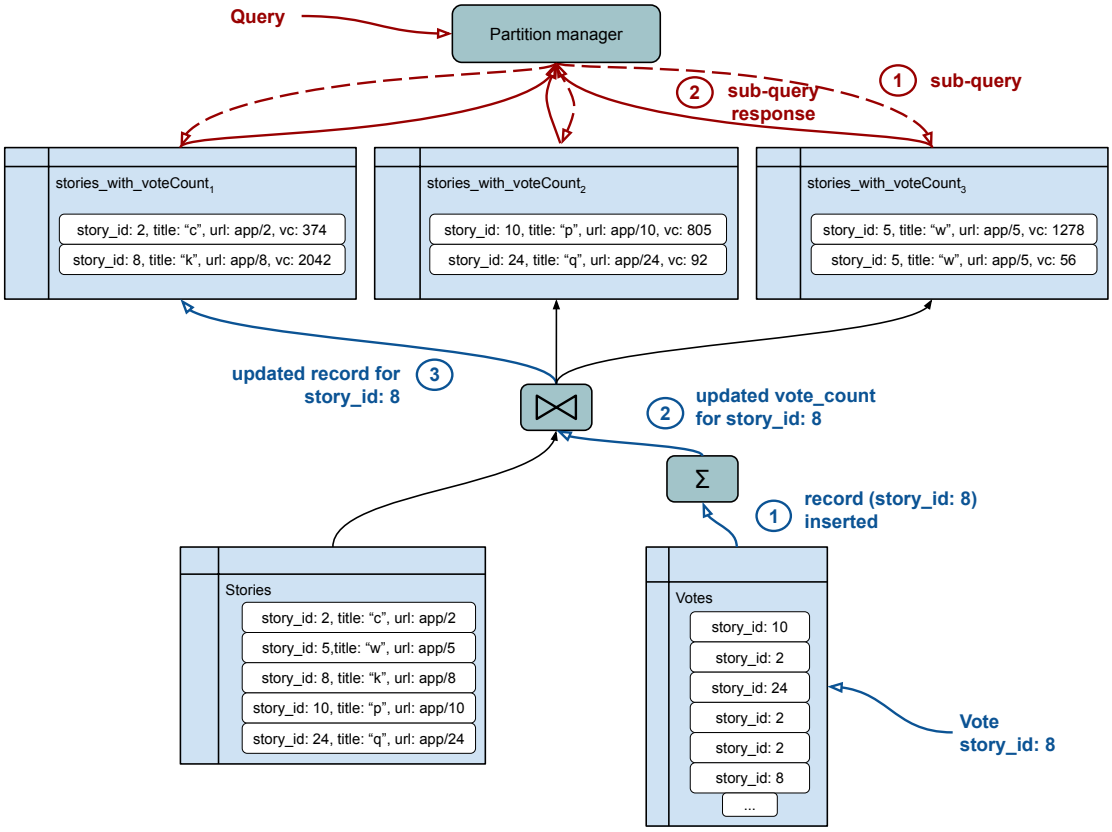


Figure 5.1: Conceptual depiction of the news aggregator application query processing architecture.

Figure 5.1 depicts a simplified version of the application's query processing architecture. An aggregation and a join operator incrementally compute the *storiesWithVoteCount* view as new stories and votes are added. The view is partitioned among multiple view partitions. A partition manager is used to manage access to the view partitions.

We use this example as a reference for discussing our observations on the semantics of query processing systems.

Breaking down query processing in basic primitives.
First, the functionality of a query processing system can be divided into three parts:

- The execution of relational algebra operations. In the news aggregator example the component that calculate the *vote_count* per story, and perform the join with the stories table are relational operators. Other examples of relational operators are selections and projections.
- Maintaining derived state structures. The materialized view partitions in the example consists derived state structures. Other examples include secondary indexes and caches.
- "Routing" functionalities. We can describe the functionality of the partition manager in the example we the term routing. This component is responsible for forwarding queries to the view's partitions and combining the return responses. There are other functionalities of query processing system component that can be categorized as routing, such as load balancing, and the federation of independent corpora.

Encapsulating query processing primitives to system components with service semantics.
Second, each of these types of functionalities can be encapsulated by an *independent system component* with microservice semantics: the component provides its functionality as a service through a "service interface"; other components can request the component's functionality by invoking its interface. In the news aggregator example, we can model the partition manager as a microservice that abstract the underlying architecture, exposing only an interface for service queries. Similarly, we can implement each view partition as an independent component that exposes an interface encapsulating the view's query method. The partition manager forwards a given query to each view partition by invoking its interface. Finally, we can also apply this to components with relational operator functionalities: We can model the

join operator as a component that provides its join operation functionality as a service. Each view partition invokes the join operator's interface using the view definition.

Next, we examine the service interface's response for the three types of query processing functionalities.

Relational operators *transform* an input set of attribute tuples, to an output set of tuples:

- A selection operator selects the input tuples for which a given condition is true, and emits them as output tuples.
- A projection operator discards a given set of attributes from the input tuples.
- A join operator receives two sets of input tuples. Its output is a set of tuples containing a combination of the attributes in the input tuples. In the example, the join operator extends the story's attributes by adding the *vote_count* attribute.
- An aggregation operator (e.g. count, sum) divides input tuples into groups based on a *grouping attribute*, and applies an *aggregation function* over the tuples of each group. In the example, the count operator groups stories by *story_id* and calculates the count of tuples for each *story_id* value.

We can unify these transformations by modeling the output of a relational operator component as a tuple of attributes.

Derived state structures accelerate read access to a set of tuple attributes, but not perform any transformations. In the news aggregator example, the view partitions pre-compute the results of the join and count operation. Therefore, the response to an invocation a view partition's interface is the same as the response of the join operator.

Routing operators either forward the responses of other components, or combine them by merging multiple input sets of tuples to a single output set.

Our third observation is that we can use a common model for the responses of all three types of query processing functionalities. The response to the invocation of a components service interface is a set of tuples of attributes. These attributes might be combinations of the attributes present in the data items of the corpus, or the results of functions applied on those attributes.

Combining the above observations, we conclude that that query processing functionalities belonging in the three categories described above can be encapsulated by a microservice-like component that provides its functionality through an interface. The response to an invocation of this interface can be modeled as is a set of data items, each structured as a tuple of attributes.

The query processing component's input port.
Our forth observation is related to the *input* required for each query processing functionality type:

- Indexes and materialized views expose a second interface for being updated to reflect changes to the corpus. In the news aggregator example, a materialized view partition receives *information about changes in the join operation result* from the join operator. The following types of changes can occur in the join operation result:
  - A new story has been created.
  - The vote count for an existing story has changed.
  - A story has been deleted.
- A cache, in the case of a cache miss, forwards the given query to a different component, and then receives the corresponding query result as input.
- As discussed above, relational operators receive a set of attributes tuples as input. Unary operators, such as selection, receive one set of tuples, and binary operators, such as joins, receive two sets.
- By definition, because of their functionality, routing components, receive the output of other components as input. The view partition manager in the example receives as input the responses of the index partitions to a given query.

We can categorize these inputs in two types:

- An index or materialized view requires as input information about *modifications* to the corpus. Moreover, the *scope* of the input required by these components *spans their entire operation lifetime*. In the example, index partitions need to *continuously* receive information about story creation and deletion of stories and votes, in order to remain up-to-date with the corpus.
- Every other query processing functionality requires as input a information about the *state* of the corpus. Moreover, contrary to indexes and materialized views, the *scope* of this input is a *single session*. In the example, the index partition manager receives as input the responses of view partitions for a specific query. Each query corresponds to a distinct set of input, tied to the specific query.

Our forth observation is that we can unify the two different *type* and *scope* input semantics. First, we can represent both input scopes using streaming semantics. The input required by an an index or materialized view can be viewed as a continuous stream of records encoding information about modifications to the corpus. The input required by other components can be viewed as a stream of records that encode parts of the state of the corpus. Using streaming semantics, we can couple the scope of an input to the lifetime of a stream. The input to a view partition in the example is *long-running* stream, the lifetime of which spans the lifetime of the partition. An input to the partition manager is a stream that spans the processing of a specific query.

Second, we can represent both input types by structuring stream records as *deltas*. A delta consists of a base state, and information encoding a modification to that state. Applying this to the news aggregator example, we can represent the input to a view partition as a record with a base state part, and a modification part. The base state part is a tuple (*story_id*, *title*, *url*, *vote_count*). The modification part may indicate that the story identified by the base state part is a newly created story, that this story has been deleted, or that this is an modified state of this story.

By omitting the modification part, the delta can also represent a part of corpus' state. Applying this to the news aggregator example, we can represent the input to the partition manager for a given query with the same record structure. In this case, the modification part is empty. The base state part represents the state of story that matches a given query.

Using this observation, we extend the query processing component described above with a "port" for receiving an input stream consisting of records structured as deltas.

The relation between service interface and input port.
Our fifth observation is that the service interface and the input port of the query processing component, are not independent from one another. In the example, the input that the partition manager receives from an view partition is the direct result of the partition manager invoking the interface of the partition. Moreover, the stream of modifications that a view partition receives, can be viewed as the result of an invocation of the join operators interface by the partition.

Guided by this observation, we extend the semantics of the query processing component as follows: The response to an invocation of the component's service interface is a stream of records structured as deltas.

To enable this connection between service interface and input port, we distinguish two types of interface invocations. A "state" invocation is evaluated *a single time* against a specific state of the corpus and produces a set of *state* results. In the example, the invocation of a view partition's interface by the partition manager is a "state" invocation. It describes a query that is evaluated once against the state of the partition, and produces a set of results. A "persistent" invocation is a long-running request that is re-evaluated upon each modification of the corpus. In the example, the invocation of the join operator's interface by a partition is a "persistent" invocation. It describes a query that is repeatedly re-evaluated each time join operator receives an input.

The second connection between input port and service interface is that the response to an interface invocation might be the result of a transformation over the input. This directly maps to the functionality of relational operators. In the example, the join operator emits a new output record as a result of receiving an input record, either from the stories table, or the count operator.

Expanding on this, we distinguish two mechanisms that query processing components use provide their exposed service:

- A derived state component process requests by reading from its state (index, cache or materialized view). This is the case for index partitions in the example.
- Relational operator and routing components provide their functionality by performing a transformation over their inputs.

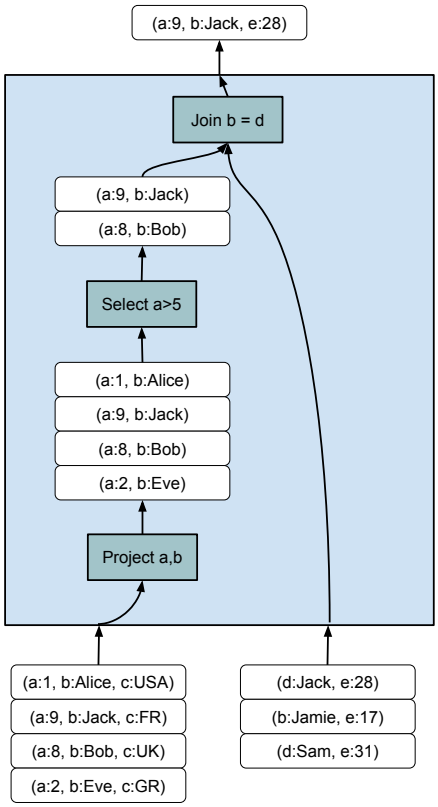


Figure 5.2: Example of achieving higher-order functionality through composition of basic functionalities.

Composition of query processing components.
Our sixth observation is that while each individual component encapsulates a basic query processing functionality, we can implement higher-level functionalities be *composing* components. This is depicted in Figure 5.2. Components with basic functionalities, such as selection, project and join, are composed to execute a more complex task.

This a common technique used in existing query engines: the execution of relational algebra expressions is represented as a dataflow computation performed by a set of operators, each performing a basic functionality, connected in a directed acyclic graph [62, 107]. The graph's components are relational operators that receive one or more input streams, perform an operation on them, and emit an output stream. Graph nodes are connected so that the output stream of one operator is the input stream of another.

We can expand the composition property to also include relational operator components as well as derived state and routing components. In the new aggregator example, we compose derived state components (view partitions), along with a routing component (partition manager) to construct a partitioned materialized view.

Conclusion.
Combining the above, we define a *system component* that combines properties of a streaming operator and a microservice. Similarly to a streaming operator, the component receives one or more input streams, performs a computation over these streams, and emits an output stream. Similarly to a microservice, the component provides its functionality as a service through an interface. The response to an invocation of this interface has streaming semantics. Combining the two, any input stream is initiated as the response to the invocation of a component's service interface. We call this component a Query Processing Unit (QPU).

An individual QPU provides a basic query processing functionality, such as filtering an input stream based on a given condition, caching query results, or routing query requests to index partitions. Higher-order query processing functionalities are achieved by *composing* multiple QPUs.

The mechanism that enables composition is that a query processing unit is able to invoke the service interface of other units. In that way, given a query, a QPU can perform "sub-queries" to other units, and retrieve the corresponding responses as input. It can then perform a computation on these "partial results" in order to compute the response to the given query.

We refer to the QPU's service interface as *query interface*, and an invocation of that interface as a *query request*.

The query processing unit is the building block of the proposed query processing system architecture. A query processing system is a directed acyclic graph with QPUs as its vertices. The graph's vertices represent potential query request — response stream relations. An edge from Q_a to Q_b indicates that Q_a can send query requests to — and therefore establish input streams from — Q_b. We call Q_b a *downstream connection* of Q_a, and a query request sent from Q_a to Q_b a *downstream query*. The corpus is located the leaves of the graph (nodes with only incoming edges), and client queries enter the graph through its root nodes (nodes with outgoing edges).

The *computation model* of query execution emerges from the semantics of the query processing unit. When a QPU at the root of the graph receives a query request, is has two mechanisms available for processing it:

- In the case of a stateful QPU it can process the given query, by reading from its internal state. For example, this is the case in a secondary index or materialized view component.
- It can establish input streams by invoking the interface one or more downstream connections, and perform a computation over these streams. For example, given a query, a QPU that implements a filter operator invokes the interface of a downstream connection. This initiates an input stream. The operator filters the input stream based on the condition specified by the query, and emits the item that satisfy the condition as the response stream.

These mechanisms are not mutually exclusive. A cache QPU first reads from its internal state. In case the results of the given query are not present in the cache, the QPU forwards the the query to a downstream connection, and receives the results as an input stream.

When the downstream query mechanism is used, this process is repeated at each QPU that receives a query request. A client query, submitted at the root of the graph, progressively spawns sub-queries that flow downwards through graph, defining a *query execution sub-graph*. Query results flow upwards through the edges of this sub-graph, and are progressively combined to produce the final result.

Query execution is therefore *decentralized*: Each QPU takes *local decisions* based on its functionality and a given query request. The result of these local decisions is a distributed computation that performs query execution.

5.2 The query processing unit: a building block for composable query processing architectures

5.2.1 The Query Processing Unit component model

In the previous section, we discussed our insights for unifying the semantics of the different types of query processing tasks under a common component that can be used as the building block of a modular query processing architecture. Building on a set of observations, we described an overview of the semantics of this component, called the query processing unit.

In this section, we formalize the query processing unit specification. To achieve that, we introduce the query processing unit *component model* (QPU Model for short). The QPU Model defines the set of common properties that every query processing unit conforms to. This includes the QPU's query interface, and the query request – response stream semantics. Using object-oriented programming terminology, the QPU Model can be viewed as an *abstract class* that defines a set of method signatures, but not their implementation.

We user the QPU model as a *template* for defining *QPU classes* with specific functionalities. For example "cache", "join", and "index partition manager" classes provide different query processing functionalities, but all conform to the properties of the QPU Mode. QPU classes can be viewed as classes that implement the QPU abstract class.

Finally, *QPU instances* can be viewed as specific instances of a QPU class, with a specific configuration. For example, different instances of the "secondary index" class can be used to implement the partitions of a partitioned index, or indexes on different attributes. As another example, a specific materialized view is implemented by an instance of the "materialized view" class, with the view's definition as a configuration parameter.

In the rest of this thesis, we use the terms query processing unit, QPU, and unit interchangeably to refer to QPU instances.

The query processing unit component model defines a *system component* with the following properties.

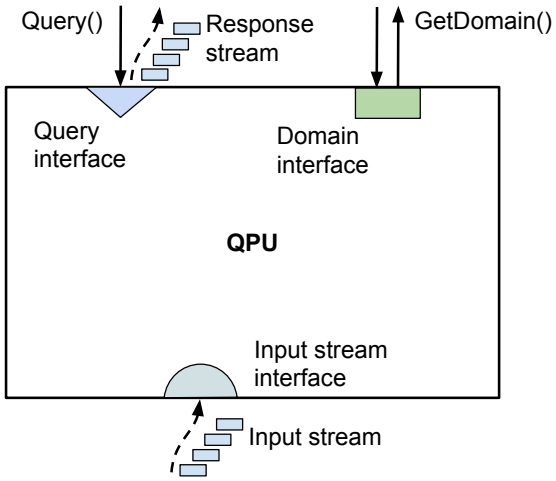


Figure 5.3: A conceptual depiction of the QPU model interfaces.

5.2.1.1 Query, input stream, and domain interfaces.

The principal characteristic of the QPU component is that is exposes three interfaces for communicating with other QPU instances. Figure 5.3 shows a conceptual depiction of the query processing unit model's interfaces.

Query interface. Query processing units expose an interface for receiving query requests. This interface is common among all QPU classes so that a QPU can send query requests to other QPUs, regardless of their class. This is the mechanism with which QPUs interoperate to perform query processing tasks, and forms the basis of the query processing system's computation model (§5.3.2).

Each QPU instance specializes in a subset of the queries that can be expressed by the interface's query language. This set of queries depends on its class, its configuration (§5.2.1.2), and the topology of the parts of the QPU graph is is connected to. For example, a QPU may only serve queries about a specific database table, while a Join class QPU will specifically serve join queries using partial results. The set of queries that a QPU instance can process is the query interface's *domain*.

The QPU's query interface has *streaming semantics*. An invocation of the query interface initiates a stream between the QPU that sent the query request and the one that received it; query results are returned through that stream; each stream record is a result of the query. This allows the QPU model to bridge one-off and long-running queries. We present the query interface in more detail in Section 5.2.2.1.

Input stream interface. Issuing a query request initiates a stream of query results. The input stream interface is the interface through which the query processing unit receives this input stream. This interface is not open: a QPU cannot receive an input stream not associated with a query that it has issued.

Domain interface. As presented above, each QPU instance specializes in a specific set of queries, which is called it's query interface domain. In order for QPUs to collaborate during query processing by issuing queries to one another, a unit must have information about the domains of its downstream connections. The domain interface allows domain information to be disseminated through the QPU graph: a unit can use the domain interface of one of its downstream connections to retrieve its domain. We present the domain interface in more detail in Section 7.1.

5.2.1.2 Configuration

Upon initialization, each QPU is given a set of configuration parameters. Configuration parameters can be categorized in:

- Class-specific configuration, which includes parameters such as cache size or the definition of a materialized view.
- Topology configuration, which consists of endpoints of a QPU's downstream connections in the QPU graph.

5.2.1.3 Local graph view

A QPU maintains information about the query interface domain of each of its downstream connections. It uses this information to: (1) validate if it can process a given query, and (2) generate downstream

sub-queries in order to retrieve partial results required for processing a given query. The QPU implements its local graph view as a data structure, called the domain tree (§5.3.3.2).

5.2.1.4 Query Processing State.

Each QPU maintains *internal* state, that is accessible only by that QPU.

We distinguish the QPU state in three parts according to its functionality. Moreover, QPUs that have downstream connections maintain information about these connections, which is used for generating downstream query requests. We call this part of the state *local graph view*. Finally, query processing units that implement derived state structures and QPUs that store intermediate query processing results, for example in streaming join computations, maintain *query processing state*.

5.2.1.5 Initialization, query processing, and input stream processor methods.

The functionality of a query processing unit can be modeled using three methods:

- The *initialization method* (Section 5.2.2.3), which is executed when the QPU is initialized.
- The *query processing method* (Section 5.2.2.4), which is executed for each query request received by the QPU, and is responsible for processing the query and sending results to the response stream.
- The *input stream processor method* (stream processor method for short) (Section 5.2.2.5), which is executed for each record received through an input stream.

The QPU model defines the signatures of these methods, and each QPU class provides implementations for them. The functionality of a QPU class is defined by the implementations that the class provides for the three QPU methods.

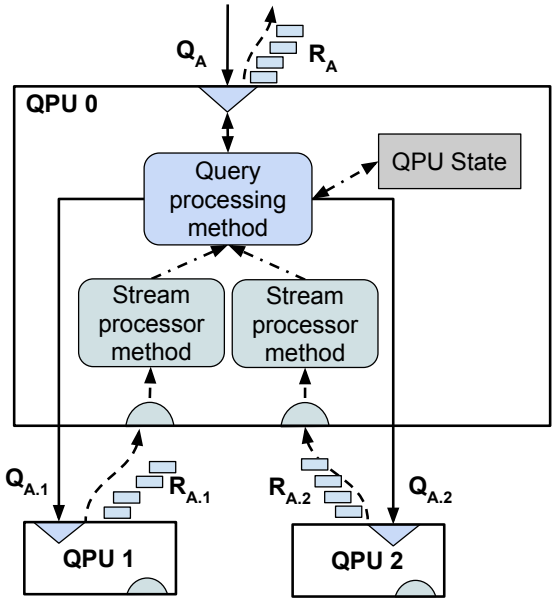


Figure 5.4: QPU composition.

Figure 5.4 shows a conceptual depiction of how the QPU model enables composition. When the QPU's query API is called, an output stream (R_A) is established between the unit and the sender, and the unit's query processing method is invoked for the given query. The query processing method can read the QPU's state, and can perform downstream queries to other units. For each downstream query, a corresponding stream is established ($Q_{A.1}$ and $Q_{A.2}$). When a record is received from one of the streams, the QPU's stream processor method is invoked. Each invocation of the stream processor method processes a received record, and returns the result to the query processing method. Upon receiving a result from a stream processor method, the query processing method can potentially write to the QPU's state and/or emit a record to the output stream.

5.2.2 Query Processing Unit component model specification

In this section we present the detailed specification of the query processing unit component model.

5.2.2.1 Query interface

Query processing units expose an interface for receiving query requests:

$$Query(QueryRequest) \rightarrow QueryResponse$$

- *QueryRequest* specifies a projection, a predicate on the data items' *attributes*, and a *timestamp* or *time interval*.
- *QueryResponse* is a stream containing the query's results.

Query response.
QueryResponse is a stream with the following structure:

$$QueryResponse = [StreamRecord]^*$$

where

$$StreamRecord =$$
$$(DataItemID,\ [(AttributeName,\ AttributeValue_{old},\ AttributeValue_{new})],\ Timestamp)$$

Query types.
We categorize queries in three types: snapshot, interval and hybrid queries.

Snapshot queries. A snapshot query is an "one-time" query that is evaluated on a snapshot of the corpus data, specified by a timestamp, and returns the data items that match the query predicate.

Given a query Q, specifying a projection $Proj_Q$, a predicate $Pred_Q$, and a timestamp t_Q, a snapshot query is evaluated on a snapshot of the corpus that contains the effects of all updates with $timestamp < t$. For each data item d that satisfies $Pred$, *QueryResponse* contains a *StreamRecord* of the form:

$$(DataItemID,\ [(AttributeName,\ null,\ AttributeValue_{new})],\ Timestamp)$$

The attributes contained in *StreamRecord* are specified by $Proj_Q$.

This represents the state of the data item $DataItemID$ at timestamp $Timestamp$. Each element of [($AttributeName$, $AttributeValue_{old}$, $AttributeValue_{new}$)] indicates that the data item is associated with an attribute $AttributeName$ with the value $AttributeValue_{new}$

Interval queries. An interval query is a "long-running" (*persistent*) query. It is akin to subscribing to a query: Each time an update to the corpus causes a result to be added or removed from the query's results, or an existing result to be modified, the system notifies the subscriber about the change.

Given a query Q, specifying a projection $Proj_Q$, a predicate $Pred_Q$, and an interval $[t_1, t_2)$, a *StreamRecord* is added to *QueryResponse* each time one of the following is true:

- An update creates a data item d, and d satisfies Q
- An update deletes a data item d satisfied Q before deletion.
- An update modifies an existing data item d, causing to either start satisfying Q, stop satisfying Q, or continue satisfying Q but with some of the attributes in $Proj_Q$ having been modified.

This specification represents the following cases:

- An update modifies a data item so that its state before the update is applied does not satisfy $Pred$, but its state after the update is applied satisfies $Pred$. In that case, the data item needs to be added to the posting list of an index entry, or to the results of a materialized view.
- An update modifies a data item so that its state before the update is applied satisfies $Pred$, but its state after the update is applied does not satisfy $Pred$. In that case, the data item needs to be removed from the posting list of an index entry, or to the results of a materialized view.
- An update modifies a data item, both its state before the update is applied and after satisfies $Pred$, and the data item's state stored as part of the index entry of the materialized view (because of $Proj$) is modified.

In this case of interval queries, *StreamRecord* has the form:

$$(DataItemID,\ [(AttributeName,\ AttributeValue_{old},\ AttributeValue_{new})],\ Timestamp)$$

This represents an update to the the data item $DataItemID$, with timestamp $Timestamp$; Each elements of [($AttributeName$, $AttributeValue_{old}$, $AttributeValue_{new}$)] indicates that the update modified the value of attribute $AttributeName$ from $AttributeValue_{old}$ to $AttributeValue_{new}$.

Hybrid queries. A hybrid query is a combination of a snapshot and an interval query. It is used in order to avoid missing updates in the common case in which a snapshot query is directly followed by an interval query.

Given a query Q, specifying a projection $Proj_Q$, a predicate $Pred_Q$, and an interval $[t_1, t_2)$, a hybrid query is equivalent to executing two queries in parallel: A snapshot query with timestamp t_1, and an interval query with interval $[t_1, t_2)$. In this *QueryResponse* contains a mix of both types of *StreamRecord*.

Response stream mode.
The query response stream can be either synchronous or asynchronous. In a synchronous stream, the sender blocks until the stream records has been received and processed by the receiver.

Query Language.
The interface supports a query language with an SQL-like syntax. The query language supports point and range queries, logical operators, aggregation functions, and joins. We argue that these is no inherent limitation in the QPU model that prevents it from supporting a more complex query language (for example supporting nested queries). However, we consider additional functionalities to be out of the scope of this work. We believe this query language can effectively demonstrate that the QPU model can be used as building block for constructing fully-functional query processing systems.

The QPU model's query language has following syntax:

```
QueryRequest          ::=  SELECT SelectExpression
                           FROM TableExpression
                           WHERE PredicateExpression
                           INTERVAL IntervalExpression
                           [ GROUP BY attributeName ]
                           [ ORDER BY OrderByExpression { ASC | DESC } ]
                           { SYNC | ASYNC }

SelectExpression      ::=  ALL | ProjectionExpression

ProjectionExpression  ::=  ProjectionItem |
                           ProjectionItem, ProjectionExpression |

ProjectionItem        ::=  COUNT(attributeName) |
                           SUM(attributeName) |
                           AVG(attributeName) |
                           MAX(attributeName) |
                           MIN(attributeName) |
                           attributeName

TableExpression       ::=  tableName JOIN tableName On
                           tableName.attributeName = tableName.
                               attributeName |
                           tableName

PredicateExpression   ::=  ( PredicateExpression ) |
                           PredicateExpression OR
                              PredicateExpression |
                           PredicateExpression AND PredicateExpression |
                           NOT PredicateExpression |

                           Term Op Term

Term                  ::=  attributeName | Value

Op                    ::=  > | >= | < | <= | = | !=

Value                 ::=  stringValue | floatValue | intValue |
                           dateTimeValue | timestampValue

IntervalExpression    ::=  FROM TimestampTerm [ TO TimestampTerm ]

TimestampTerm         ::=  SYSTEM START | LATEST | timestampValue
```

```
OrderByExpression      ::=  attributeName
```

Listing 5.2: Query language of the QPU's query interface. TableExpression corresponds to an equi join.

The `INTERVAL` syntax is used to specify the query type. Queries without a `TO` part are snapshot queries; queries with a `TO` are interval or hybrid queries. Queries with `LATEST` keyword in the `FROM` part are interval queries, while queries with with a specific timestamp (or the `SYSTEM START` keyword) are hybrid queries. The `SYSTEM START` keyword is used to indicate the earliest available timestamp.

The `SYNC` or `ASYNC` is used to specify the response stream mode.

In practice, each QPU instance specialized in a subset of the queries that can be expressed by this query language, according to the functionality supported by its class. For example, a QPU class that implements a join operator can perform a join over two input streams, but cannot evaluate a $PredicateExpression$, or perform an SUM on an attribute of the join result. This can be achieved by connecting "join", "filter", and "aggregator" QPUs. We present in detail these QPU classes in Section 5.2.4, and how they are composed to provide complex query processing functionalities in Section 5.3.

Moreover, instances of a certain class may support different parts of the query language according to their configuration. For example, two instances of a filter operator QPU class may support predicate queries for different tables.

5.2.2.2 Query processing state

Query processing units that encapsulate derived state structures such as indexes, materialized views and caches, store these data structures in the part of their state that we call *query processing state*. Additionally, streaming operator QPUs such as joins use this part of the state to store intermediate state.

We model the query processing state as set of key-value pairs, ordered by key:

$$(Key,\ [State],)$$

where

$$State = (DataItemID,\ [(AttributeName,\ AttributeValue)],\ Timestamp)$$

Listing 5.3 defines the QPU's query processing states using pseudocode:

```
type AttributeName string

type AttributeValue union {
  string
  float
  int
  dateTime
  timestamp
}

type State {
  dataItemID  string
  attributes  [(AttributeName, AttributeValue)]
  ts          timestamp
}

class QueryProcessingState
  function Get(Key_low, Key_high, Timestamp_low, Timestamp_high) [(Key, [
      State])]
  function Put(Key, State)
```

Listing 5.3: Pseudocode for the QPU's query processing state

- *Put* modifies the value of the query processing state's entry for the given key. It can be used with a non-existing key to create a new entry.
- *Get* retrieves the query processing state entries with $Key_{low} < key \leq Key_{high}$. For each entry, the updates with $Timestamp_{low} < Timestamp \leq Timestamp_{high}$ retrieved.

Essentially, the QPU's query processing state is versioned. In that way, QPU's can process queries about any timestamp that they have received updates for.

5.2.2.3 Initialization method

Each QPU invokes an initialization method when it starts its execution.

```
type DownstreamConnection {
  endpoint string
  domain DomainTree
}

function Init(QPState, [DownstreamConn])
```

Listing 5.4: Initialization method signature

Listing 5.4 shows the initialization method's signature. *ProcessQuery* receives as arguments:

- *QPState*, which is a handler that enables *Init* to read from and write to QPU's query processing state.
- A list of *DownstreamConn*, each consisting of the endpoint of a downstream connection and a data structure representing the connection's query interface domain (Section 5.3.3.2). *Init* can therefore send downstream query requests.

5.2.2.4 Query processing method

The query processing method is responsible for processing a single query. It spawns a result stream and sends the results back to the requestor over that stream.

For each query q being processed, the QPU initiates an output stream, R_q, and executes the query processing method, QPF_q. QPF_q is responsible for emitting records to R_q. Moreover, if QPF_q initiates input streams I_{q-1}, I_{q-2}, ..., then QPF_q is responsible for handing the return values of stream processor methods for records received through I_{q-i}.

```
function ProcessQuery(QueryRequest, QPState, [DownstreamConn],
    ResponseStream)
```

Listing 5.5: Query processing method signature

Listing 5.5 shows the query processing method's signature. In addition to *QPState* and [*DownstreamConn*], *ProcessQuery* receives:

- *QueryRequest*, which represents the received query request.
- *ResponseStream*, a handler it can use to emit result to the output stream.

The query processing method is executed for each received query request, therefore query processing unit can run multiple instances of the query processing method in parallel.

5.2.2.5 Input stream processor method

The input stream processor method is responsible for processing a record received through an input stream. It can read from and write to the query processing unit state, and a (potentially empty) list of *StreamRecord* to the query processing method.

```
function ProcessInputRecord(StreamRecord, QueryRequest, QPState)
          returns [StreamRecord]
```

Listing 5.6: Stream processor method signature

Listing 5.6 shows the query processing method's signature. An instance of the stream processor method is executed for each record received through an input stream.

5.2.3 Stream semantics

In this section we describe more detail the semantics of the stream connections between query processing units.

Apart from *data records* (updates), *control records* can also be sent through a stream. Although data records can be sent only in one direction ("upstream"), control records can be sent in both directions. Types of control records include acknowledgements, heartbeats, and resend requests.

Sending a data record can be either *synchronous* or *asynchronous*. A synchronous send operation blocks until an acknowledgement for the data record is received.

Finally, we assume that stream connections do not drop, re-order, or duplicate messages; however messages may be arbitrarily delayed. These guarantees are not based on assumptions about the transport layer, but as we discuss in Chapter 8 are implemented by query processing units on top of potentially unreliable connections.

5.2.4 QPU classes

As described in the previous section, the query processing unit component model has the role of "template", defining unified semantics that every QPU conforms with. This ensures that QPU instance with different functionalities (classes) and configurations can be interconnected and can interoperate to perform query processing tasks.

A QPU class is an *instantiation* of the query processing unit model: it defines the implementations of the **initialization**, **query processing** and **stream processor** methods.

In this section, we present a categorization of QPU classes according to their general characteristics, and demonstrate some examples of QPU classes. We present additional QPU classes in Chapters 6, 7, 8.

We categorize QPU classes in three groups, according to their general characteristics:

- **Relational operator classes**. Classes in this group encapsulate *streaming relational operators*. A relational operator QPU receives one or more input data streams, and perform a transformation over these streams, and emits an output stream.

  Every input stream of relational operator QPU is the output stream of another QPU, and has been establish as a response to a query request.

  Examples of QPU classes in this group are:

  - **Filter:** A selection QPU receives records from an input stream, and emits at its output those records satisfy a given condition.
  - **Join**: A join QPU encapsulates a streaming join operator. It receives records from two input streams, and performs a join operation over them, according to parameters specified by a query request. The join QPU initiates input streams by sending the appropriate query requests to its downstream connections, and emits ths join operation result as its output stream.
  - **Aggregator classes**: Aggregator QPU classes encapsulate aggregation functions such as count, sum, average, and min or max. An aggregator QPU performs a streaming aggregation over an input stream; It emits a record at its output stream for each input record that changes the computed aggregation value.

- **Derived state classes**. Classes in this group encapsulate derived state structures, such as indexes, caches, and materialized views.

  This group includes the following classes:

  - **Secondary index and Materialized view:** Secondary index and materialized view QPUs support both incremental updates and re-evaluation from the corpus. For the rest of this chapter, we focus on the case in which a secondary index or materialized view is initially evaluated when created, and then incrementally updated.

    A secondary index (or materialized view) QPU receives its index or view definition as a configuration parameter. Its initialization method establishes an input stream by sending an *interval* query (a query without an upper bound timestamp) to its downstream connection: In that way, the QPU effectively *subscribes to notifications* for updates to the corpus. For each record received through the input stream, the stream processor method updates the query processing state accordingly. When a query request is received, the query processing method computes the results by reading from the query processing state, and emits them to the output stream.
  - **Cache:** When receiving a query request, a cache QPU's query processing method first determines if the query result is stored in the query processing state. If yes, the method retrieves the corresponding query results and emits them at the output stream. Alternatively, it sends a query request at the QPU's downstream connection, with the same query. The cache QPU's input stream processor method stores each record received as a response that request, and then returns it to the query processing method which emits it to the output stream.

- **Routing classes**. Classes in this group encapsulate query processing functionalities that can be characterized as "query routing". This includes coordinating access to partitioned or replicated derived indexes and materialized views, or load balancing.

  Examples of classes in this group include:

  - **Partition manager:** A partition manager QPU is responsible for coordinating access secondary index or materialized view partitions, encapsulated by the corresponding QPU classes.

    A partition manager QPU has downstream connections to a set of QPUs representing partitions. When a query request is received, the QPU's query processing method determines which partitions need to be contacted, generated the corresponding queries and sends them as downstream query requests. To enable this, the partition manager QPU maintains information about the partitioning scheme and the portion of the partitioned space that corresponds to each of its downstream connections connections at its query processing capabilities spaces The unit then merges the input streams and emits the result as its output stream.

- **Load balancer and replica manager:** QPUs of these classes have similar functionalities with the partition manager class. Given a query, the query processing method of a load balancer or replica manager QPU selects the most suitable among the QPU's downstream connections according to a certain criterion (defined by QPU's class and configuration), forwards the given query to that connection, and then forwards the resulting input stream to the output stream.

- **Corpus driver classes**. Classes in this group are responsible for connecting the QPU graph with the corpus. QPUs of these classes do not support downstream connections to other QPUs. Because of their functionality, corpus driver QPUs are only used as leaves of the QPU graph. In fact, as we explain in Section 5.3.1.1, every leaf node of the QPU can only be a corpus driver QPU.

  When a query request is received, the query processing method of a corpus driver QPU uses the interface and mechanisms of they corpus database in order to generate and emit the output stream that corresponds to the given query. This can include reading from the database that stores the corpus, subscribing to update notifications, or performing queries.

  The corpus driver classes act as a wrappers for the corpus, exposing a common interface and semantics to the QPU graph, independent of how the corpus is stored and accessed. As a result, corpus driver classes are database-specific.

  QPU-based query processing systems can be compatible with any corpus database — or, more generally, source of updates — for which there is a corresponding corpus driver class.

5.2.4.1 QPU class case studies

Filter

Algorithms 1 and 2 show the query processing and stream processor methods respectively for the selection QPU class using pseudocode.

Algorithm 1 Selection QPU class query processing method

```
function ProcessQuery(queryReq, qpState, [downstreamConn], outputStream)
    conn = get endpoint from [downstreamConn]      ▷ We assume one downstream connection
    inStream = forward queryReq to conn
    while not at end of this the stream do
        inRecord = inStream.Recv()
        result = ProcessInputRecord(inRecord, queryReq, qpState)
        if result is not empty then
            outStream.Send(result)
        end if
    end while
end function
```

Algorithm 2 Selection QPU class stream processor method signature

```
function ProcessInputRecord(inRecord, queryReq, qpState)
    if inRecord satisfies queryReq then
        return inRecord
    else
        return []
    end if
end function
```

Secondary index

Algorithms 4, 5, and 3 show the initialization, query processing, and stream processor methods respectively for the secondary index QPU class using pseudocode.

5.3 QPU-based query processing systems

5.3.1 Query processing system architecture

A *QPU-based* query processing system is a directed acyclic graph (DAG) with query processing unit instances as graph nodes. Edges represent potential query request - response stream relations between

Algorithm 3 Secondary index QPU class stream processor method

```
function ProcessInputRecord(inRecord, queryReq, qpState)
    (dataItemID, [attributeDelta], timestamp) = parse inRecord
    for (attrName, attrValOld, attrValNew) in [AttributeDelta] do
        if attrValOld is not null then
            indexTerm = build index term from attrName and attrValOld
            postingList = qpState.Get(indexTerm, indexTerm)
            postingList' = remove dataItemID from postingList
            pState.Put(indexTerm, postingList')
        end if
        if attrValNew is not null then
            indexTerm = build index term from attrName and attrValNew
            postingList = qpState.Get(indexTerm, indexTerm)
            postingList' = add dataItemID from postingList
            pState.Put(indexTerm, postingList')
        end if
    end for
end function
```

Algorithm 4 Secondary index QPU class initialization method

```
function Init(qpState, [downstreamConn])
    conn = get endpoint from [downstreamConn]      ▷ We assume one downstream connection
    queryReq = generate interval query
    inStream = send queryReq to conn
    while not at the end of the stream do
        inRecord = inStream.Recv()
        ProcessInputRecord(inRecord, null, qpState)
    end while
end function
```

Algorithm 5 Secondary index QPU class query processing method

```
function ProcessQuery(queryReq, qpState, [downstreamConn], outputStream)
    [indexTermL, indexTermB = build terms for reading secondary index from queryReq
    result = qpState.Get(indexTermL, indexTermB)
    outStream.Send(result)
end function
```

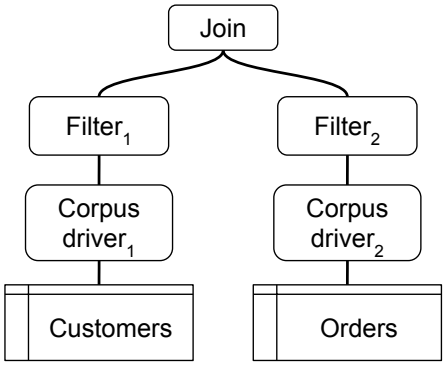


Figure 5.5: QPU graph example

QPUs: a directed edge from QPU_a to QPU_b indicates that QPU_a is able to send query requests to QPU_b. A query request sent from QPU_a to QPU_b initiates a stream connection between QPU_a and QPU_b; QPU_b emits query results through that stream.

The corpus is represented by the graph's leaf nodes (nodes with no outgoing connections). Queries enter the QPU graph through root nodes (nodes with no incoming connections).

The capabilities of a QPU-based query processing system are emergent from the functionalities of the QPUs it is composed of, as well as the graph topology. For example, consider the QPU graph depicted in Figure 5.5, where:

```
TABLE Customers(CustomerID, Email, Address)
TABLE Orders(OrderID, CustomerID, OrderStatus, OrderDate)
```

Listing 5.7: Customer and Orders tables attributes

- *Corpus driver*$_1$ can process queries with the form:

  ```
  SELECT SelectExpression from Customers INTERVAL IntervalExpression
  ```

  where *SelectExpression* can contains one or more attributes of the *Customers* table.
- *Corpus driver*$_2$ can process queries with the form:

  ```
  SELECT SelectExpression FROM Orders INTERVAL IntervalExpression
  ```

  where *SelectExpression* contains one or more attributes of the *Orders* table.
- *Filter*$_1$ can send downstream queries to *DB driver*$_1$ and filter the resulting input stream based on a given attribute predicate. Therefore, *Filter*$_1$ can process queries of the form:

  ```
  SELECT SelectExpression FROM Customers
  WHERE PredicateExpression
  INTERVAL IntervalExpression
  ```

  where *SelectExpression* and *PredicateExpression* contain attributes of the *Customers* table.
- Similarly, *Filter*$_2$ supports queries of the form:

  ```
  SELECT SelectExpression FROM Orders
  WHERE PredicateExpression
  INTERVAL IntervalExpression
  ```

  where *SelectExpression* and *PredicateExpression* contain attributes of the *Orders* table.
- *Join* can send downstream queries to *Filter*$_1$ and *Filter*$_2$, and join the two input streams based on a given attribute. Since the only attribute the two table in the example can be joined on is *CustomerID*, *Join* can process queries of the form:

  ```
  SELECT SelectExpression
  FROM Customers JOIN Orders ON Customers.CustomerID = KnowledgeBase.
      CustomerID
  WHERE PredicateExpression
  INTERVAL IntervalExpression
  ```

  where *SelectExpression* and *PredicateExpression* contain attributes of both tables.

In Section 5.3.2 we describe how a query such as the aforementioned is processed by a QPU-based query processing system. Moreover, in Section 5.3.3.2 we describe how the set of queries a query processing unit can process are represented and used by other QPUs.

5.3.1.1 QPU-graph topology rules

A QPU DAG cannot have any arbitrary topology. This is because each QPU class has specific requirements about: (1) number of the QPU's downstream connections, and (2) the query interface domain of these downstream connections. From these *class-specific topology requirements*, certain higher-level graph topology rules emerge:

- Any QPU graph mush have corpus driver QPUs as its leaf nodes. Corpus drivers generate the initial streams that all other QPUs build on.
- Most QPUs in the relational operator and derived state groups must have a single downstream connection. Exception are relational operator QPUs that by definition operate on more that one input streams, such as join QPUs.
- A materialized view QPU must have a downstream connection that:
  - Can process the query that is the view's definition.
  - Supports *interval queries* on that query.

  This is because, the materialized view QPU needs to receive an input stream of updates that correspond to changes in the result of that query, in order to incrementally update the materialized view.
- Similarly, a secondary index QPU must have a downstream connection that can provide a stream of updates on the attribute that QPU is configured to index.
- A cache QPU must have a single downstream connection, which can be any other QPU.
- Similarly, a selection QPU can have a downstream connection to any QPU.
- A partition manager QPU can have one or more downstream connections. It is not a requirement that downstream connections are partitions of a derived state structure. It is required that some combination of query interface domains of these connections result in the domain of the partition manager QPU. In practice, each connection is responsible for a distinct part of the partition manager QPU's domain.
- A load balancer QPU can have one or more downstream connections; The QPU is responsible for performing load balancing on the queries that belong in the *intersection* of the query interface domains of its downstream connection. Because of that, the intersection of the query interface domains of the downstream connection of the load balancer QPU must be non empty.

The above list is inherently non-exhaustive: any QPU class that can be defined using the QPU model may have additional requirements.

5.3.2 Query execution

The query execution computations of QPU-based query processing systems are *fully decentralized*. The distributed computations directly emerge from the initialization, query processing and stream processor methods of the QPUs the query processing system consists of. In order to describe the characteristics of query execution in QPU graphs, we first describe the execution of the query

```
Q = SELECT OrderID, OrderStatus, Email
FROM Customers JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE OrderStatus != "shipped" AND OrderDate < 2020-09-14
INTERVAL FROM LATEST TO LATEST
```

by the QPU graph of Figure 5.5.

When a *Join* receives a query request for Q, its query processing method generates two downstream queries:

```
Q1 = SELECT CustomerID, Email
FROM Customers
INTERVAL FROM LATEST TO LATEST

Q2 = SELECT CustomerID, OrderStatus
FROM Orders
WHERE OrderStatus != "shipped" AND OrderDate < 2020-09-14
INTERVAL FROM LATEST TO LATEST
```

Join sends a query request for $Q1$ to $Filter_1$, and a query request for $Q2$ to $Filter_2$.

When a $Filter_1$ receives $Q1$, its query processing method generates the query and sends it as downstream query request to $Corpus\ driver_1$:

```
Q3 = SELECT CustomerID, Email
FROM Customers
INTERVAL FROM LATEST TO LATEST
```

Similarly, $Filter_2$ generates and sends to $Corpus\ driver_2$ the following query:

```
Q4 = SELECT CustomerID, OrderStatus
FROM Orders
INTERVAL FROM LATEST TO LATEST
```

$Corpus\ driver_1$ reads from $Customers$ and generates an output stream that for each data item $d \in Customers$ contains the most recent update. Similarly, $Corpus\ driver_2$ generates an output stream for $Orders$. $Filter_2$ emits at its output stream only the input stream updates for which

```
OrderStatus != "shipped" AND OrderDate < 2020-09-14
```

is true. Because $Q1$'s $PredicateExpression$ is empty, $Filter_1$'s output stream is identical to its input stream. Finally, $Join$ receives the results of $Q1$ and $Q2$ as input streams; its query processing method join updates from the two streams where

```
Customers.CustomerID = Orders.CustomerID
```

and emits the results at its output stream.

More generally, given a query q, the query processing method of a QPU Q_0 may either process q by reading from its query processing state — for example in the case of a materialized view QPU — or generate and send query requests to some of its downstream connections, Q_1, Q_2 .., in order to initialize input streams required for processing the query. In both cases, the QPU computed the results of q and emits them at the output stream that corresponds to q.

The same process is performed at each of the downstream connections of Q_0, and their downstream connection respectively, propagating through the QPU graph from its root to its leaves. This creates a *query execution sub-graph* composed of the QPUs that participate in the processing of q. When a QPU can process a received query request without sending downstream query requests, it becomes a leaf node of the query execution sub-graph of q. QPU classes that become query execution sub-graph leaf nodes are corpus driver QPUs, and derived state QPUs. Leaf nodes produce output streams that are then received as input stream at their "parent" nodes. Progressively, each non-leaf QPU in the query execution sub-graph receives an input stream for each query request sent, and produces itself an output stream. The output stream of Q_0 contains the results of the initial query, q. We call this type computation in a QPU DAG, *query execution mode*.

QPU-based query processing systems execute a similar type of computations for incrementally updating secondary indexes and materialized views. We call this type computation in a QPU DAG, *state maintenance mode*. There are the following differences between query execution and state maintenance mode:

- Query execution is performed in response to a client query (by the query processing method of the QPU that receives the query) while state maintenance is performed in response to the initialization of a secondary index or materialized view QPU (by the initialization method of the corresponding QPU).
- The goal of query execution is to process a client query, while the goal state maintenance is to establish a long running stream of notification for corpus updates that will be used to incrementally update a derived state data structure.
- The root of a query execution sub-graph is one of the QPU DAG's root nodes, while the root of a state maintenance sub-graph is a derived state QPU that might be a root or an internal node of the graph.

State maintenance can be configured to be performed either synchronously or asynchronously, depending on the type of send operation used for sending update records. Synchronous state maintenance is achieved by configuring all QPUs.

Based on the above description, the computations run by a QPU-based processing system can be characterized as *bi-directional dataflow*. For a given query or state maintenance execution, query requests flow downwards through the QPU graph, defining an execution sub-graph. Response streams flow upwards through that sub-graph, each stream corresponding to an edge defined by a query request.

Finally, as described in Section 5.2.2, a QPU can process multiple queries in parallel, executing an instance of its query processing method and having an output stream for each query request being processed. Therefore, multiple different query execution and state maintenance sub-graph can co-exist in parallel in the same QPU DAG.

5.3.3 Query execution data structures

So far in this chapter we have presented a high level description for certain parts of the query processing unit's functionality. In this section we present this parts in details. More specifically, we present the data structure that encodes the QPUs' query interface domain, and describe how this structure is used by the query processing method to generate downstream queries for a given query.

5.3.3.1 Query parse tree

The first step of a QPU's query processing method is to parse the given query from a string to a form that can be used for query processing computations. This form is a *parse tree*.

A parse tree is constructed by applying the QPU query language's syntax rules to a given query string. More specifically, a parse tree is composed to two types of nodes:

- **Atoms**, which include keywords of the query language ($SELECT$, $FROM$ etc.), identifiers (such as table and attribute names), constants, operators and tokens. Atoms are leaf nodes of the parse tree.
- **Syntactic categories**, which are constructs built from atoms or other syntactic categories following the query language's syntax rules. Syntactic categories are internal nodes of the parse tree.

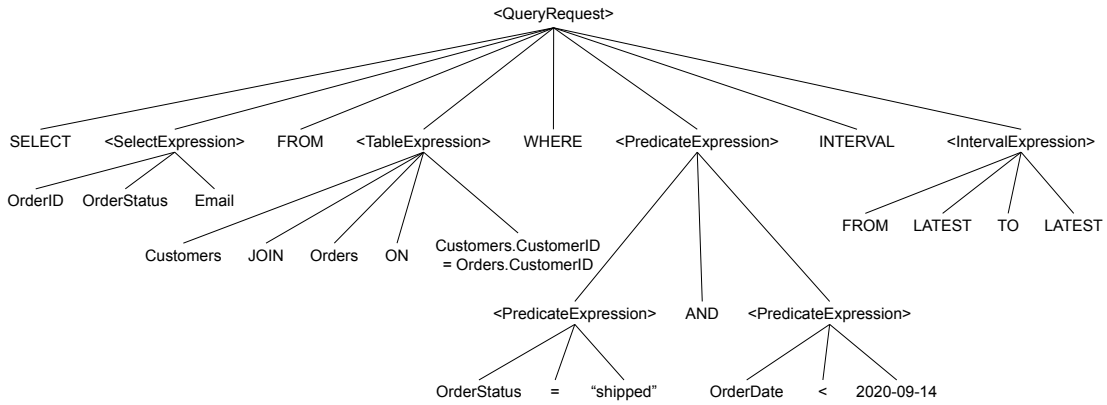


Figure 5.6: Query parse tree for the query in Listing 5.8.

For example, the parse tree for the following query is depicted in Figure 5.6:

```
SELECT OrderID, OrderStatus, Email
FROM Customers JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE OrderStatus != "shipped" AND OrderDate < 2020-09-14
INTERVAL FROM LATEST TO LATEST
```

Listing 5.8: Query used to demonstrate the query parse tree data structure.

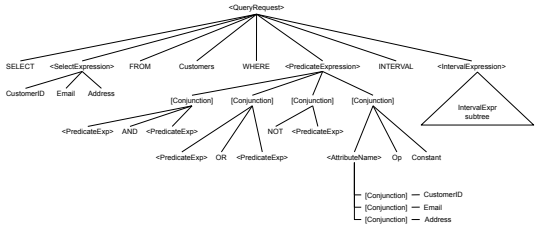
5.3.3.2 Domain tree

In this section, we present the domain tree data structure. The domain tree represents a query processing unit's query interface domain, i.e. the set of queries it can process. Because each query can be represented as a query parse tree, a domain tree is constructed by merging the query parse trees of the queries that in the QPU's query interface domain. Each QPU maintains a domain tree for each of its downstream connections. The QPU's local graph view consists of this collection of domains trees. Moreover, each unit constructs its own domain tree based on the domains trees of its connections, and its functionality. The domain tree is an extension of the query parse tree, which we presented in the previous section. It uses an additional type of tree node, called **conjunction**. A conjunction node represents the *different potential sub-trees* that a syntactic category node can be evaluated to. Essentially, a conjunction node expresses the different branches of a syntax rule.

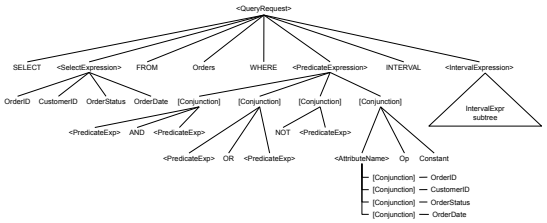
Query processing units use a straightforward algorithm based on transformation and merge rules construct their domain trees. This algorithm is specific to each QPU class:

- A Corpus driver unit constructs its domain tree based on the schema of the corpus table it is responsible for. More specifically, it 1) uses the table's name as TableExpression node, 2) creates a SelectExpression subtree according to the table's attributes, and 3) constructs the IntervalExpression sub-tree according to its configuration.

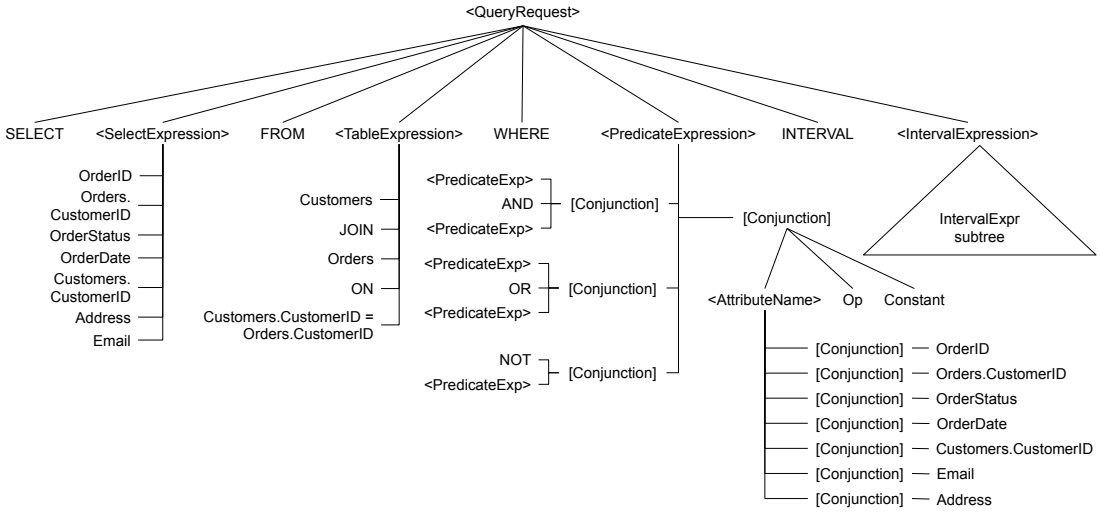
- A Filter QPU constructs its domain tree by extending the PredicateExpression node of the domain tree of its downstream connection, according to the attributes in the SelectExpression node (Figures 5.7a and 5.7b)
- A join QPU merges the two domain trees of its connections as follows. It creates a new TableExpression subtree, using the TableExpressions node of the input domain trees, and the join syntax rule (Listing 5.2), and merges the PredicateExpression subtrees of the the input domain trees using a straightforward tree merge algorithm (Figure 5.7c).
- A Partition Manager unit constructs its domain tree by combining the trees of the PredicateExpression subtrees of its connections. For example, consider the case of a partition manager QPU connected to two index QPUs. The first index QPU is responsible for index entries in the interval $[V_A, V_B)$, and the second for $[V_B, V_C)$. The Partition Manager's domain tree will represent the interval $[V_A, V_C)$.



(a) Domain tree for $filter_1$ QPU in Figure 5.5.



(b) Domain tree for $filter_2$ QPU in Figure 5.5.



(c) Domain tree for *join* QPU in Figure 5.5.

Figure 5.7: Domain trees for the QPU graph in Figure 5.5.

QPU trees have two functionalities:

- A query processing unit uses its own domain tree in order to determine whether it can process a given query. More specifically, it checks if the query's parse is a subtree of the QPU's domain tree, and, therefore, if the query's parse tree belongs in the set of queries represented by the QPU tree (algorithm 6).
- A query processing unit uses the domain trees of its downstream connections in order to generate downstream queries. Given a query q with parse tree PT_q, received a query processing unit Q that has a downstream connections Q_d with domain tree DCT_{Qd}, the parse tree of the downstream query to be sent from Q to Q_d for q is the **intersection** of DT_{Qd} and PT_q (algorithm 7).

5.4 Discussion

5.4.1 QPU-based query processing systems and ad-hoc queries

.

Algorithm 6 Algorithm for check if a query can be processed

```
function CanProcessQuery(queryReq, DomainTree)
    queryParseTree = parse queryReq
    return isSubTree(DomainTree, queryReq)
end function
```

Algorithm 7 Algorithm for generating downstream queries

```
function GenerateDownstreamQuery(queryReq, downstreamConn)
    queryParseTree = parse queryReq
    domainTree = get domain tree of downstreamConn
    downStreamQueryParseTree = intersection(domainTree, queryParseTree)
    downStreamQueryReq = generate query from parse tree
    return downStreamQueryReq
end function
```

The goal of the QPU-based design pattern is to enable *workload-driven design*. A QPU graph is designed based a predefined set of query patterns. As a result, a QPU graph does not support any ad-hoc query, but rather *the set of queries it has been designed for*. More specifically, as described in Section 5.3.3.2, a QPU graph supports the set of queries represent by the domain trees of its root nodes. Supporting additional query patterns, not supported by a certain QPU graph requires designing and deploying a new QPU graph.

5.4.2 Query processing unit consistency semantics

The query engine architecture presented maintains derived state, including indexes and materialized views, in a system that is separate to the data storage tier. This creates two considerations about the consistency of query results: the consistency between corpus and derived state, and the consistency between different indexes or materialized views, or shards of an index/view, maintained by the query engines. We discuss former in this section, and outline the latter as a direction for future work (§10.1.3).

We refer to a stateful QPU as *internally consistent* in time t, if its state has been updated according to all updates with timestamp $\leq t$, and no updates with timestamp $> t$. Our design ensures that 1) if a query q with a timestamp t_q is issued to a stateful QPU Q, and Q has received and processed an update with timestamp $\geq t_q$, then Q can evaluate q on an internally consistent snapshot of its state with timestamp t_q, and 2) any Q can evaluate any subsequent query on an internally consistent snapshot of its state with timestamp $\geq t_q$. Essentially, this property ensures that any state snapshot of a stateful QPU reflects a (potentially stale) snapshot of the corpus data.

The query processing unit design achieves that as follows:

- We assume that updates the data storage tier propagates updates to the QPU graph in-order, according to their timestamps.
- The stream connection between two QPUs ensures that streams records are not lost or re-ordered.
- Given a stream of input records with increasing timestamps, relational operator QPUs emits result records that also have increasing timestamps (potentially a subset of the input timestamps).

5.5 Conclusion

This chapter presented the design of a modular and composable query engine architecture. It introduced the Query Processing Unit, presented its specification, and described how it can be used as a building block for constructing query engine architectures.

Chapter 6

Case studies

Having presented an approach for constructing query processing architectures through assembly-based modularity, in this chapter, we demonstrate its flexibility by applying it to a number of use cases and applications. More specifically:

- We demonstrate how both a document and a term-partitioned secondary index can be implemented using QPU-based architectures (Section 6.1). In addition, we use this case study to describe in detail the construction and functionality of QPU graph, including the configuration of query processing units, and the queries sent between units during initialization and query processing.
- We examine a state-of-the-art approach to providing federated metadata search over data spread across multiple private and public cloud storage platforms, and propose a decentralized approach that places index entries closer to the corpus in order to improve freshness (Section 6.2). Moreover, we present a QPU architecture that implements partial index replication. This approach provides fine-grained control over the placement of index entries: Heavily queried index entries are placed closer to the users, while heavily updated entries are placed closer to the corpus.
- We examine a read-heavy news aggregator application, and propose a QPU-based query processing system for maintaining pre-computed state in order to improve the application's performance, while simplifying its code (Section 6.3). To achieve that, we present a QPU that implements partial materialization in materialized views: Only materialized view entries expected to be requested by the application are materialized. This reduces the view's memory footprint as well as the volume communication required for keeping view entries up-to-date with the corpus. Furthermore, we demonstrate how a QPU-based architecture can be distributed between data center and edge nodes, and use partial materialization to ensure that only heavily queried materialized view entries are placed at the edge.

6.1 Flexible secondary index partitioning

As described in Section 3.2.4.2, the two main index partitioning schemes are:

- **Partitioning by document**. Each index partition is responsible for the data items of a certain corpus partition, and is co-located in the same node as that corpus partition.
- **Partitioning by term**. Each index partition is responsible for a partition of the *value space* of the indexed attribute; The placement of index partitions is independent from the placement of corpus partitions.

Experimental comparison of the two approaches [54, 76] has shown that there is no "one-size-fits-all" approach to secondary index partitioning. Rather, each approach caters to different needs. More specifically, partitioning by document is more suitable for:

- Workloads with low selectivity queries, that return large result sets.
- Skewed distributions, in which a large number of data items correspond to a few attribute values.
- Write-intensive workloads that require low write latency.

On the other hand, partitioning by term is more suitable for:

- Large-scale systems with a large number of corpus partitions.
- High selectivity query workloads.
- Less skewed value distributions.

Because of that, applications can benefit from partitioning schemes adjusted to their data and workload characteristics.

In most existing query processing systems, the choice of index partitioning scheme is made during the system's design, and is not configurable by applications. For example, MongoDB [127], Cassandra [90] and Riak [18] only use the partitioning by document approach, while HBase [20] uses the partitioning by term approach.

In this section, we demonstrate the flexibility of the QPU-based query processing architecture by showing how it can be used to be used to express both index partitioning schemes. More specifically, we present a QPU architecture that implements a document-partitioned index, and one that implements a term-partitioned index.

The design of the QPU architectures that we present is based on observations about the properties of the two index partitioning schemes. We categorize these observations using the derived state read and write path framework presented in Section 4.1:

- In the partitioning by document scheme, the index **write path** is *local*: An index partition receives updates only from the corpus partition it is co-located with. On the other hand, in the partitioning by term scheme, the write path involves a many-to-one relationship: An index partition receives from all corpus partitions updates that belong in the value space it is responsible for.
- In the partitioning by document scheme, the **read path** is a *broadcast* operation, while partitioning by term only index partitions with relevant index entries are involved in processing a given query.

We present QPU architectures for the two index partitioning approaches using the example that we introduced in Section 4.1. The corpus is composed of a set of image files. Each image is identified by a primary key, and is associated with a set of user-defined tags. The corpus is partitioned using a hash of the primary key as the partitioning key. An application needs to create a secondary index on the *predominantColor* tag, which can be assigned values in the range [#000000, $\#FFFFFF$].

Figures 6.1 and 6.2 show the QPU graphs for a document-partitioned index and a term-partitioned index respectively, and their placement across system nodes. For simplicity we assume that the number of corpus partitions is equal to the number of system nodes, and that a corpus partition is placed on each node.

The two architectures have a number of general characteristic in common. In both architectures, a corpus driver QPU is placed on each node, and is responsible for the corresponding corpus partition. It provides the QPU graph with access to the data items in that corpus partition, as well as the updates performed on those data items. In addition, an index QPU is used to represent each index partition; A partition manager QPU is connected to all index partitions, and is responsible for coordinating query access to them.

In Sections 6.1.1 and 6.1.2 we describe how the two QPU architecture achieve the write and read path properties described above:

6.1.1 Write path

The write path properties described above are achieved through (1) the QPU graph topology and the placement of graph vertices across system nodes, and (2) the configuration of the index QPUs (index partitions).

6.1.1.1 Graph topology and placement

Partitioning by document. In the partitioning by document approach, we place an index QPU on each system node, and connect it to the corresponding corpus driver QPU. As a result, each index partition has access corpus partition it is co-located with, and therefore constructs and maintains an index containing the data items that belong in the corpus partition.

Partitioning by term. In the partitioning by term approach, we connect each corpus driver QPU to a selection QPU, and connect each index QPU to all Selection units. A selection QPU forwards each update to the relevant index partition, based on the attribute value in the update. We describe in detail how this is achieved in the following section.

6.1.1.2 Index partition configuration and graph initialization

When the QPU graph for one of the partitioned index architectures is initialized, the initialization method of each index QPU sends a persistent query to the QPU's downstream connection, in order to establish an input stream of updates (the initialization method of the index QPU is presented in Section 5.2.4.1). This downstream query is generated by the initialization method according to the QPU's configuration. More specifically, each index QPU is configured with an attribute (*predominantColor* in this example), and an

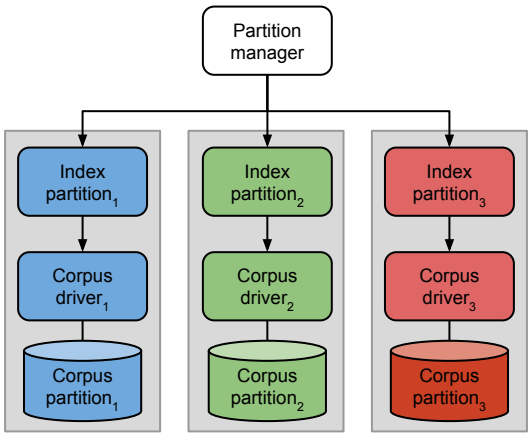


Figure 6.1: QPU architecture for a document-partitioned secondary index.

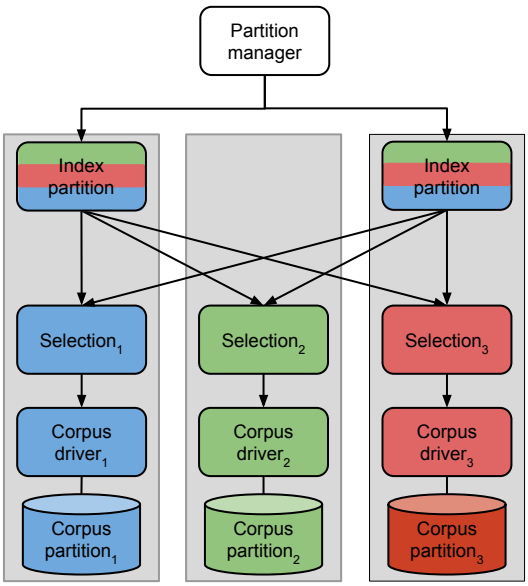


Figure 6.2: QPU architecture for a term-partitioned secondary index.

interval of values of that attribute. The index partition is responsible for index entries that correspond to attribute values in the specified interval.

In the partitioning by document approach, each index partition is configured to be responsible for the entire attribute value space, which in the photo album example is [#000000, $\#FFFFFF$]. As a result, upon initialization, each index partition sends the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
INTERVAL FROM LATEST
```

Listing 6.1: Query sent during initialization from each index partition to the corresponding corpus driver QPU, in the QPU graph shown in Figure 6.1.

to the corpus driver QPU it is connected to. This initiates a stream between each corpus driver QPU and the corresponding index QPU The Corpus Driver initially sends a snapshot of the corresponding corpus partition to the index QPU, and then sends a stream record for each corpus update.

Conversely, in the partitioning by term approach, each index partition is responsible for a non-overlapping subset of the attribute value space. A simplified version of the configuration of the index partitions of the term-partitioned index QPU architecture (Figure 6.2) is depicted below:

```
{
  // other configuration
     parameters
  "indexConfiguration": {
    "table": "photoAlbum",
    "attribute":
        "predominantColor",
    "lower_bound": #000000,
    "upper_bound": #7FFFFF
  }
}
```

Listing 6.2: Configuration of the *index partition*$_1$ in Figure 6.2

```
{
  // other configuration
  // parameters
  "indexConfiguration": {
    "table": "photoAlbum",
    "attribute":
        "predominantColor",
    "lower_bound": #7FFFFF,
    "upper_bound": #FFFFFF
  }
}
```

Listing 6.3: Configuration of *index partition*$_2$ in Figure 6.2

As a result, upon initialization, *index partition*$_1$ sends the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
WHERE predominantColor >= #000000 AND predominantColor < #7FFFFF
```

```
INTERVAL FROM LATEST
```

Listing 6.4: Query sent during initialization from *index partition*$_1$ to to each selection QPU in the QPU graph shown in Figure 6.2

to each downstream selection QPU, and *index partition*$_2$ sends the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
WHERE predominantColor >= #7FFFFF AND predominantColor <= #FFFFFF
INTERVAL FROM LATEST
```

Listing 6.5: Query sent during initialization from *index partition*$_2$ to to each selection QPU in the QPU graph shown in Figure 6.2

to each downstream selection QPU.
As a result of receiving one the above queries, each selection QPU sends to each downstream corpus driver QPU the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
INTERVAL FROM LATEST
```

Listing 6.6: Query sent during initialization from each selection QPU each corpus driver QPU in the QPU graph shown in Figure 6.2

As a result, each selection QPU receives an input stream from the corresponding corpus driver QPU, for each of the two above queries, and filters the received streams according to the predicate specified by each query. Therefore, *index partition*$_1$ receives from all three corpus partitions first a snapshot and then updates corresponding to the *predominantColor* in the interval [#000000, $\#7FFFFF$). Similarly, *index partition*$_2$ receives stream records corresponding to the *predominantColor* in the interval [$\#7FFFFF$, $\#FFFFFF$).

For example, given a update that inserts to the corpus an image file with *predominantColor* = #613930, assigned to to *corpus partition*$_2$, *Corpus Driver*$_2$ sends to *Selection*$_2$ a record encoding that update. This input record matches the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
WHERE predominantColor >= #000000 AND predominantColor < #7FFFFF
INTERVAL FROM LATEST
```

Listing 6.7: Query that retrieves image files from the *photoAlbum* table based the value of the *predominantColor* attribute

and thus the *Selection*$_2$ forwards the record only to *index partition*$_1$.

6.1.2 Read path

In both QPU architectures, the partition manager QPU is connected to all index QPUs. As described in Section 5.3.3.2, given a query, the Partition Manager's query processing method determines which downstream connections need to be contacted. It generates the corresponding downstream queries using the domain trees of its downstream connections.

More specifically, given a query, the query processing method, generates a query parse tree, and performs an intersection between the query parse tree and the domain tree of each of its downstream connections. The result of each intersection operation is a query parse tree of the downstream query to be sent to the corresponding connection. If the intersection result is empty, then the QPU does not send a downstream query to that connection.

Partitioning by document. The domain tree for an index partition in the document-partitioned index QPU architecture is shown in Figure 6.3. As describe above, in the document-partitioned index, each index partition is responsible for the entire attribute value space of the data items of one corpus partition. The domain tree of every index partition is thus are the same as the one depicted in Figure 6.3. Moreover, because this tree corresponds to the entire attribute value domain, the intersection with any query parse tree is an identity function: Performing an intersection between that domain tree a query parse tree leaves the parse tree unchanged.

Therefore, given a query, the partition manager QPU forwards the same query to each of its downstream connections. This implements the required read path behavior.

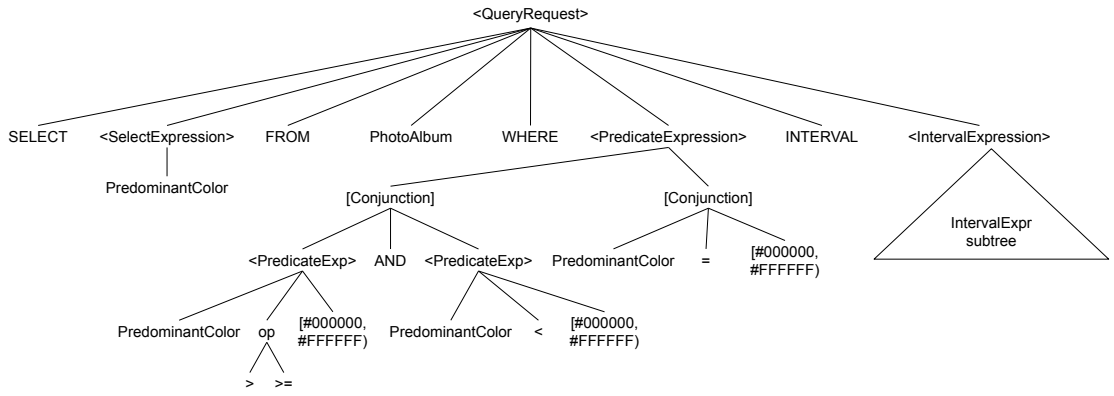


Figure 6.3: Domain tree of an index partition in the document-partitioned index QPU architecture.

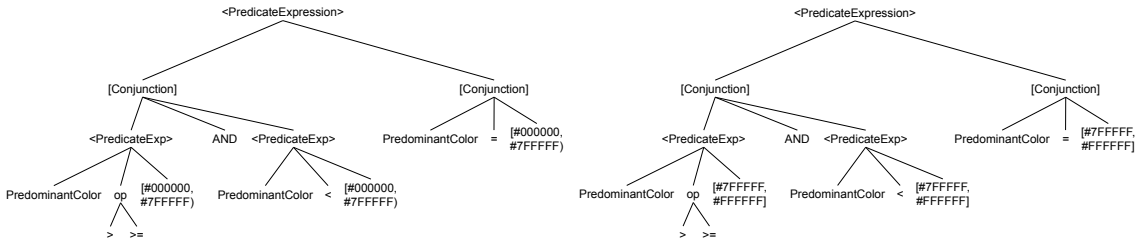


Figure 6.4: The domain tree for *index partition*$_1$ (6.4a) and *index partition*$_2$ (6.4b) of the term-partitioned index in Figure 6.2

Partitioning by term Figure 6.4 shows the $< predicateExpression >$ subtrees of the domain trees for *index partition*$_1$ and *index partition*$_2$ in the term-partitioned index architecture. The domain tree of *index partition*$_1$ represents the set of queries in the interval $[\#000000, \#7FFFFF)$, while the domain tree of *index partition*$_2$ represents the interval $[\#7FFFFF, \#FFFFFF]$.

We describe how the term-partitioned index architecture processes queries by running through an example query. Given the query:

```
Q = SELECT primaryKey, predominantColor
    FROM photoAlbum
    WHERE predominantColor >= #21B1FF
    AND predominantColor < #ff7b75
    INTERVAL FROM LATEST
```

Listing 6.8: Query that retrieves image files from the *photoAlbum* table based the value of the *predominantColor* attribute

the partition manager QPU calculates the intersection between the parse tree of Q the domain tree of *index partition*$_1$. This results to the downstream query:

```
Q1 = SELECT primaryKey, predominantColor
     FROM photoAlbum
     WHERE predominantColor >= #21B1FF
     AND predominantColor < #7FFFFF
     INTERVAL FROM LATEST
```

Listing 6.9: Sub-query sent from the partition manager QPU to *index partition*$_1$ in Figure 6.4 in order to process the query in Listing 6.8.

Similarly, the downstream query for *index partition*$_2$ is:

```
]
Q2 = SELECT primaryKey, predominantColor
     FROM photoAlbum
     WHERE predominantColor >= #800000
     AND predominantColor < #ff7b75
```

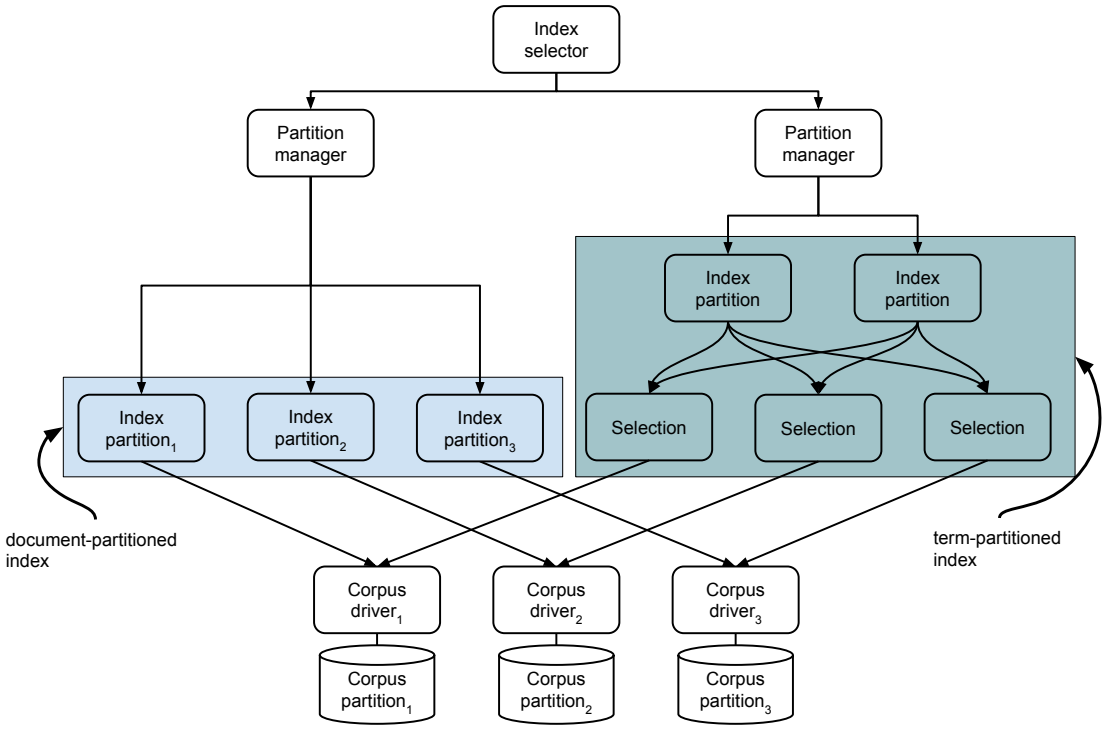


Figure 6.5: QPU architecture for a two-tiered partitioned index, composed of a document-partitioned, and a term-partitioned tier.

```
INTERVAL FROM LATEST
```

Listing 6.10: Sub-query sent from the partition manager QPU to *index partition*$_2$ in Figure 6.4 in order to process the query in Listing 6.8.

The partition manager QPU sends $Q1$ to *index partition*$_1$ and $Q2$ to *index partition*$_2$. It then merges resulting input streams, and emits the merged stream as its output stream.

In summary, in the term-partitioned architecture, the Partition Manager generates and sends downstream queries to index partitions according their corresponding value intervals, using the domain tree mechanism.

In conclusion, we have described how the QPU-based composable query processing architecture can be used to construct both document and term partitioned indexes. Using this flexibility, a database system can support both partitioning schemes and provide applications the ability to select the partitioning scheme of each indexes according the data distribution characteristics and the expected workload.

It is important to note that some existing databases, such as Amazon DynamoDB [146] and Apache Phoenix [21] support both index partitioning schemes. However, we believe that the proposed approach provides a more structured approach to the design of query processing systems, and enables additional flexibility.

For example, using the QPU-based architecture we can construct a hybrid, two-tiered partitioned secondary index, consisting of a document-partitioned tier and a term-partitioned tier, as shown in Figure 6.5. We configure the document-partitioned tier to be strongly consistent with the corpus, by configuring each connection between an index QPU and the corresponding Corpus Driver to be synchronous. We configure the term-partitioned tier to be eventually consistent with the corpus, by configuring each Index — Corpus Driver connection as asynchronous.

The document-partitioned trades consistent query results with more limited query load scalability, as each query needs to be forwarded to every index partition. The term-partitioned index trades higher query load scalability, with potentially stale query results.

Using this query processing architecture, individual queries are able to choose between the two tiers according to their requirements. To achieve this, we introduce the index selector QPU class. The index selector is responsible for managing access to the two index tiers, by forwarding each query to one of the partition manager QPUs it connected to. It selects between its connections based on an indication by the given query: Queries can select between the two partitioned index tiers using a special "control" attribute, called *tier*. The query:

```
SELECT primaryKey, predominantColor
```

```
FROM photoAlbum
WHERE predominantColor = #ff7b75
AND tier = sync
```

Listing 6.11: Query that retrieves an image file from the *photoAlbum* table based the value of its *predominantColor* attribute.

will be forwarded to the document-partitioned index, while the query:

```
SELECT primaryKey, predominantColor
FROM photoAlbum
WHERE predominantColor = #ff7b75
AND tier = async
```

Listing 6.12: Query that retrieves an image file from the *photoAlbum* table based the value of its *predominantColor* attribute.

will be forwarded to the term-partitioned index.

This architecture trades this functionality with the resources required for maintaining two index replicas.

6.2 Federated secondary attribute search for multi-cloud object storage

While most enterprise data today originate from and is stored on-premises storage systems, use cases for hybrid and multi-cloud storage are emerging in many industries. For example, in the media industry, while the creation of content in on-premises private clouds is prevalent, the use of public cloud services for content distribution and transcoding [115] is growing. Moreover, organization increasingly choose to spread their data across multiple public cloud providers in order to avoid dependence on a single provider, and improve their resilience against failures.

The advent of data distributed across multiple independent storage platforms has created the need for unified access to data across platforms. In this section, we examine Zenko [118], a multi-cloud data controller that aims to address this need. Zenko provides a common namespace over a set of distinct object storage platforms, including Amazon S3, Microsoft Azure Blob storage, and Google Cloud Storage. It allows applications to access data in multiple storage locations using the AWS S3 API [145].

We focus on Zenko's federated metadata search functionality [119]. It is common for application to mark objects with metadata tags. Zenko provides applications the ability to retrieve objects based on queries on their metadata tags, independent from storage location.

Zenko uses a warehousing approach to provide federated metadata search: It integrates metadata tags for objects stored on all storage platforms in a central *metadata store* [117, 120]. More specifically, it uses a MongoDB deployment as this metadata store. MongoDB provides the ability to create indexes on document attributes, and to retrieve documents using queries on these attributes. Zenko uses this functionality to implement metadata search: Object metadata are stored as documents in MongoDB. Queries on metadata tags are translated to MongoDB find requests.

A typical Zenko deployment consists of Zenko along with a private storage system deployed on-premises, and multiple public cloud storage platforms, each on a different data center. For this case study, we refer to each data center / storage system as a *storage location*. The primary data access method is through Zenko's API: Application communicate with Zenko to read and write objects, and Zenko is responsible for storing and retrieving each object from the corresponding storage location according to a specified policy. In addition, applications can write directly to some of the storage systems. Zenko ingests updates performed directly to an underlying storage system through a mechanism called out-of-band updates [121].

In this case study, we present an alternative approach for supporting federated metadata search on object storage platforms. More specifically, we demonstrate how the QPU-based query processing architecture can be used to construct a *decentralized* secondary index that federates data stored on multiple storage locations; The main idea is to partition the secondary index based on the storage location of each object, and place each index partition close to the corresponding data.

There are two main advantages to this approach compared to the warehousing approach:

- It ensures that the write path of the index does not require cross data center communication. This means that the system can update index partitions synchronously without the prohibitive overhead of cross data center communication. Alternatively, if index maintenance is asynchronous, the index can by updated in a more timely fashion, resulting in less stale index entries. In general, a decentralized index is better suited for applications that require up-to-date query results.
- By being distributed across multiple data centers, the query processing system can remain available in the face of a data center failure.

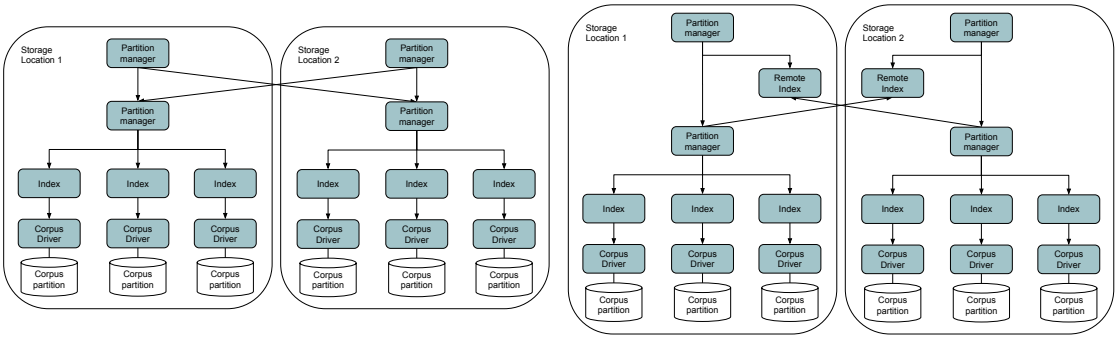


Figure 6.6: QPU architecture for a secondary index that federates two storage locations. (a) Using a Partition Manager as a root at each storage location that forwards queries to every storage location. (b) Using *partial index replicas* implemented by partial index QPUs.

We make two assumptions: The first is that storage platforms used in this case study have a common, object storage data model. Second, a corpus driver QPU is available for each storage platform. This assumptions ensure that a QPU graph can be connected with multiple underlying storage systems.

The QPU architecture for the multi-cloud index is shown in Figure 6.6 (a). We built on the partitioned index QPU architectures presented in Section 6.1. We construct a partitioned index QPU graph for each storage system, and co-locate it in the same data center as that storage system. We refer to each of these QPU graphs as an *index location*. Each index location is independent and can be constructed using either the partitioning by document or partitioning by term approach. In addition, a partition manager QPU is deployed on each storage location, and connected to all index locations. The partition manager QPUs consist the root nodes of the QPU graph. Given a query, a partition manager QPU forwards the same query to every index location, and then merges the resulting input streams. This approach is equivalent to the document-partitioned index approach.

The downside of this approach is that the index query processing system needs to forward each query to every storage location.

This can be addressed by replicating the entire index at each storage location, but that would multiply the storage and memory footprint of the index. In addition it would shift the requirement for cross data center communication to the write path; Every update would need to be sent to every other data center.

We propose an alternative approach, based on partial index replication. To achieve that, we introduce the partial index QPU class. The goal of the partial index class is to implement secondary index in which only a subset of index entries are materialized, essentially implementing a *partial replica of a secondary index*. The insight is that some index entries are accessed more frequently than others; The partial index, similarly to a cache, materializes the most recently read index entries; Additionally, it does not invalidate entries, but, similarly to an index, performs incremental updates to keep them up-to-date with the corpus. This has two benefits: It bounds the memory footprint of the index, and reduces the cross data center communication needed by the index both in the read and the write path. In the read path, queries that can be processed using materialized index entries do not need to be forwarded to another storage location. In the write path Using the remote index QPU reduces the cross data center communication required for index processing by replicating recently accessed index entries.

We use the partial index QPU to propose an improvement to the QPU architecture of Figure 6.6 (a). We deploy partial index QPU on each storage location, and connected to the index sub-graph of the partial index. Each partial index QPU works as follows. When initially deployed, it none of its entries materialized. When it receives a query, its query processing method forwards the query to its downstream connection, which in this QPU graph is an index location, as a persistent query. It stores the received entries, incrementally updates them using the input stream of updates. A difference with the index QPU is therefore that the partial index establishes an input stream for each materialized index entry, while the Index requires a single input stream.

While the partial index replication can potentially reduce the write path cross data center communication of an index replica, it might still incur significant data transfer between storage locations, in cases in which materialized index entries are updated frequently. To address this issue, we extend the partial index class with a mechanism for measuring the rate of updates to each index entry. If the update rate for a certain entry crosses a threshold specified by the QPUs configuration, then the QPU stops receiving updates for this entry and discards it.

The partial index QPU provides fine-grained placement of index entries. Index entries that are queried more heavily are replicated, and thus placed closer to clients, while index entries that are updated more heavily are placed closer to the corpus.

This technique has two benefits. First it reduces the memory and storage resources required for query

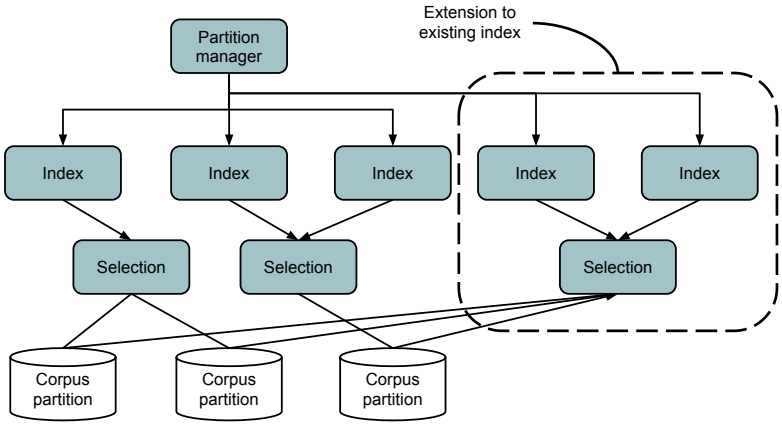


Figure 6.7: QPU architecture for a predicate-based secondary index.

processing by partially replicating index entries. Second, it reduces network resource consumption for data transfer between storage locations, as each index entry is placed at the storage location it communicates more with.

The QPU architectures described in this case demonstrate an important property of QPU-based query processing architectures: A QPU graph can have a hierarchical structure, in which lower layers are composed composed of multiple independent sub-graphs, which are then connected by higher layers. Th federated index architecture consists the partitioned index for each storage location (lower layer), connected by partition manager QPUs (higher layer). Moreover, the lower layer is transparent to the higher layer. For example, the lower layer of the federated index architecture can used either the partitioning by term or partitioning by document approach without affecting the system's functionality.

6.2.1 Predicate-based indexing

In this section, we extend the multi-cloud index case study by examining a use case inspired by one of Scality's customers. We consider the example of a media organization that operates broadcasting and streaming services. It stores media assets (video content, images) used by these services in an object storage system. As described above, objects are tagged with metadata tags; Applications use queries on metadata tags to provide recommendation and user search functionalities. The organization operates in two geographic locations, and a separate storage platform is used for each location.

The metadata search requirement of this use case are as follows: Applications on each geographic location need low latency metadata search on *local* corpus (data stored on that geographic location), as well as a *subset* of corpus in the remote location. This subset is specified by the application, and is expressed as predicates on metadata tags. In other words, for an application that operates in $location_A$, there is an "interest set" in $location_B$: a subset of the corpus $location_B$ that the application accesses frequently and needs low latency metadata search on.

While the state-of-the-art warehousing approach of constructing a central index with metadata tags from all objects on both locations can be applied to achieve these requirements, that would potentially usesignificantly more resources that required, if the interest set is small compared to the corpus.

Instead, we construct a secondary index that indexes only the interest set, by filtering the corpus based on a given predicate before indexing. Figure 6.7 shows the QPU architecture used to achieve that. We use a layer of selection QPUs between the corpus (corpus driver QPUs) and the partitioned index. The selection QPUs apply a filter on the corpus based on the query that defines the interest set, making only data items that satisfy it available to the index. We refer to this architecture, as an *predicate-based secondary index*.

A property of this architecture is that it is extensible: Additional corpus subsets can be added to the filtered index by deploying additional Selection — index QPU subgraphs, and connecting them through a partition manager QPU.

A federated index QPU architecture for this use case consists of two parts follows the same logic as the one presented in Figures 6.6 (a) and (b), with the addition that in one of the two storage locations we use a predicate-based index.

6.3 Materialized view middleware

In this case study, we examine an existing application that requires the use of pre-computed state, and study how it can benefit by using a QPU-based architecture for query processing.

Lobsters [51] is a news aggregator web application. In Lobsters, users post and comment on links (*stories* in the Lobsters terminology). In addition, users vote on stories and comments, and votes are used to rank stories.

Lobsters a is read-heavy application. Traffic data for the production deployment of Lobster, provided by Lobsters' administrators [104] 88% to 97% of the users' interactions with the application are operations tha perform reads to the application's backend. These include viewing specific stories (55%) and viewing the frontpage (30%).

In applications such as Lobsters in which read performance is important, application developers often implement mechanisms to optimize it. Lobsters, in addition to storing individual votes in a votes table, also stores per-story vote counts and story rankings as additional columns of other tables [53]. Pre-computing and storing vote counts and story rankings avoids the need re-compute them on every page load. However, the application logic needs to explicitly update the pre-computed values every time a vote is casted.

Another approach that read-heavy applications employ to avoid expensive computation in read queries is to use an in-memory key-value store, such Redis or memcached [99] as a cache to speed up common-case queries. The approach reduces load to the database, as queries can be served from the cache when the underlying records are unchanged. However, the application needs to explicitly invalidate and replace cache entries as database records change. This requires complex application-side logic, and is error-prone.

In this case study, we focus on a subset of Lobster's functionality that can de modeled as follows:

```
TABLE users (id int, username text)
TABLE stories (id int, author_id int, title text, url text);
TABLE votes (user_id int, story_id int, vote int);
```

Listing 6.13: The database schema used in this case study.

We demonstrate a QPU architecture that expresses a materialized view which pre-computes the *vote_count* for each story. The goal of this view is to facilitate the query that Lobsters executes for retrieving a requested story:

```
Q_story = SELECT id, author_id, title, url, vote_count
          FROM stories
          JOIN (
            SELECT story_id, SUM(vote) as vote_count
            FROM votes
            GROUP BY story_id
          ) view
          ON stories.id = view.story_id
```

Listing 6.14: Query that retrieves a story along with its vote count, based on its ID.

Figure 6.8 shows the QPU architecture that implements this materialized view. We pass *Q_story* as configuration to the materialized view QPU. Upon initialization, the materialized view QPU sends the following as the following downstream query to the join QPU:

```
SELECT id, author_id, title, url, vote_count
FROM stories
JOIN (
  SELECT story_id, SUM(vote) as vote_count
  FROM votes
  GROUP BY story_id
) view
ON stories.id = view.story_id
INTERVAL FROM LATEST
```

Listing 6.15: Query from the materialized view QPU to the join QPU in Figure 6.8 during initialization.

The join QPU in turn generates two downstream queries:

```
Q1 = SELECT id, author_id, title, url
     FROM stories
     INTERVAL FROM LATEST
```

Listing 6.16: Query sent by the join QPU to the *stories* corpus driver QPU in Figure 6.8 during initialization.

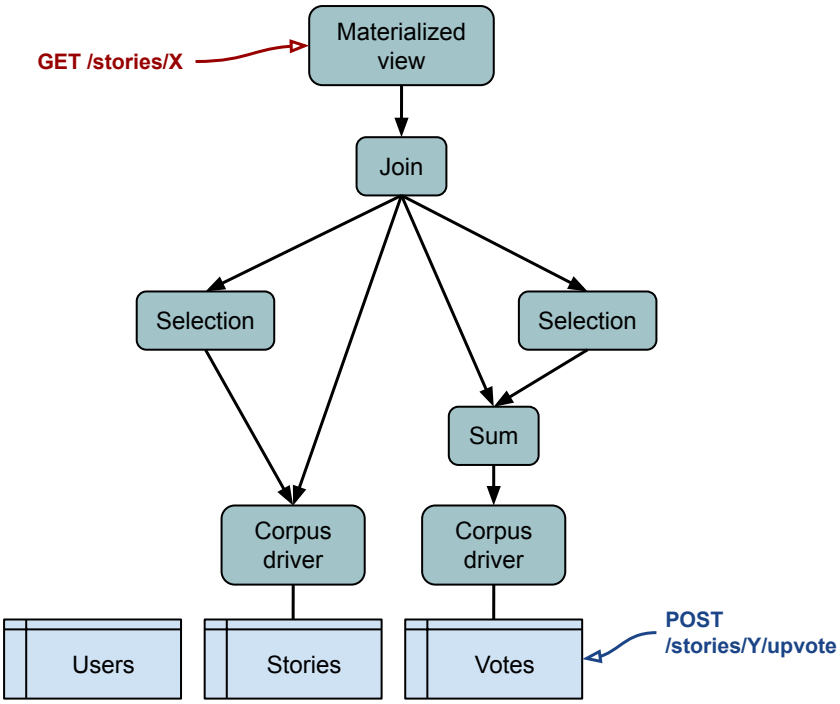


Figure 6.8: QPU architecture for the materialized view that integrates vote counts to stories.

```
Q2 = SELECT story_id, SUM(vote)
     FROM votes
     GROUP BY story_id
     INTERVAL FROM LATEST
```

Listing 6.17: Query sent by the join QPU to the sum QPU in Figure 6.8 during initialization.

It sends $Q1$ to the *stories* corpus driver QPU and $Q2$ to the sum QPU. We note that it may send the two downstream queries to the corresponding selection QPUs; The two query plans are equivalent.

When the corpus driver QPU receives $Q1$, it initiates an output stream, first sending a snapshot of all stories in the stories table, and then sending an update for each write to the table.

When the sum QPU receives $Q2$, its query processing method sends a downstream query to the *votes* corpus driver QPU in order to receive the most recent snapshot of the votes table and subscribe to subsequent writes to the table. As a result, tt receives an input stream consisting of each record in the votes table snapshot, and incrementally calculates the sum of the *vote* attribute (*vote_count*) for each distinct *story_id*. Its state is composed of (*story_id*, *vote_count*) tuples. When the sum QPU completes processing the snapshot, it emits every computed (*story_id*, *vote_count*) tuple at its output stream. In addition, the sum QPU also receives an input record for each insert to the votes table. Any update record received while still processing the snapshot, is treaded as a snapshot record; It updates corresponding *vote_count*, but does not result to an output record. For each update received after the votes table snapshot has been processed, the sum QPU updates the *vote_count* for the corresponding *story_id*, and emits the updated (*story_id*, *vote_count*) at its output stream.

Join stores intermediate state for each of input stream. When it receives an input record from one of the streams, it matches it with the corresponding stored record for the other stream, based on the *story_id* attribute.

In our implementation of this use case, the join QPU is merged with the materialized view QPU so that the system does not need to maintain state in order to perform the join operation, in addition to the materialized view state (Section 8.1.1).

In summary, this QPU architecture provides the functionality of pre-computing story votes counts in order to avoid re-computation when serving user read requests. This is equivalent to the functionality already implemented in the Lobsters application. However, using the QPU-based materialized view shifts the responsibility of updating the vote counts from the application logic, to the query processing architecture, thus simplifying the application code.

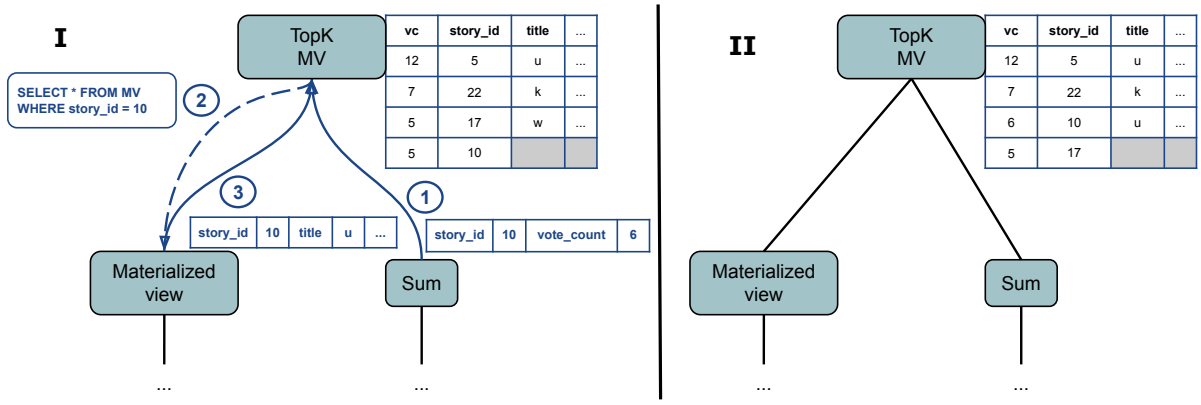


Figure 6.9: A vote triggers the materialization of an entry of the TopK materialized view.

6.3.1 Partial materialization

We expand the Lobsters case study by examining the operation of loading the application's frontpage. The frontpage consists of the K stories with the highest vote count (K being a parameter).

The Lobsters implementation uses the following query to retrieve the information required for loading the front page:

```
SELECT id, author, title, url, vote_count
FROM stories
JOIN (
  SELECT story_id, SUM(v.vote) as vote_count
        FROM votes
        GROUP BY story_id
) view
ON stories.id = view.story_id
ORDER BY vote_count DESC
LIMIT K
```

Listing 6.18: Query that retrieves the stories in the Lobsters front page.

In this section, we show how the QPU architecture defined in the previous section can be extended to provide this functionality.

To achieve that, we introduce an additional query processing unit class: the TopK materialized view (TopK-MV) QPU. The TopK-MV represents a materialized view which orders its entries based on a specified attribute. In addition, it only materializes the K entries with the highest (or lowest) values for that attribute.

We deploy a TopK-MV QPU in the Lobster application QPU architecture (Figure 6.9), and configure it as a materialized view with the same definition as the existing materialized view, storying entries of the form (*story_id*, *author_id*, *title*, *url*, *vote_count*). In addition, we configure TopK-MV QPU to order entries by *vote_count*, and materialize the K entries with the highest *vote_count*.

The TopK-MV is connected to the materialized view QPU and the sum QPU. Its initialization method sends a persistent query to the sum QPU. As a result, it first receives a snapshot containing the *vote_count* of every existing story in snapshot of the votes table; It subsequently receives an update for each change to the *vote_count* of a story. It stores the received snapshot as a list of (*vote_count*, *story_id*) entries, ordered by *vote_count*. Then, for each of the K entries with the highest *vote_count*, it sends a downstream query to the materialized view QPU in order to retrieve the remaining attributes of the corresponding story. Therefore, the TopK-MV *materializes* a bounded number of entries, based on a given criterion.

Furthermore, the TopK-MV QPU continues receiving updates from the sum QPU, and updating its ordered list of (*vote_count*, *story_id*) entries. When the *vote_count* of a non-materialized entry becomes one of the top-K, the unit's stream processor method discards the materialized entry with the lowest *vote_count*, and triggers the materialization of that entry.

The Top-K QPU uses the technique of *partial materialization* in order to bound the size of pre-computed state. This is a well known concept for materialized views in database systems [150, 152], and has been used by Noria [62] in the context of data-flow systems.

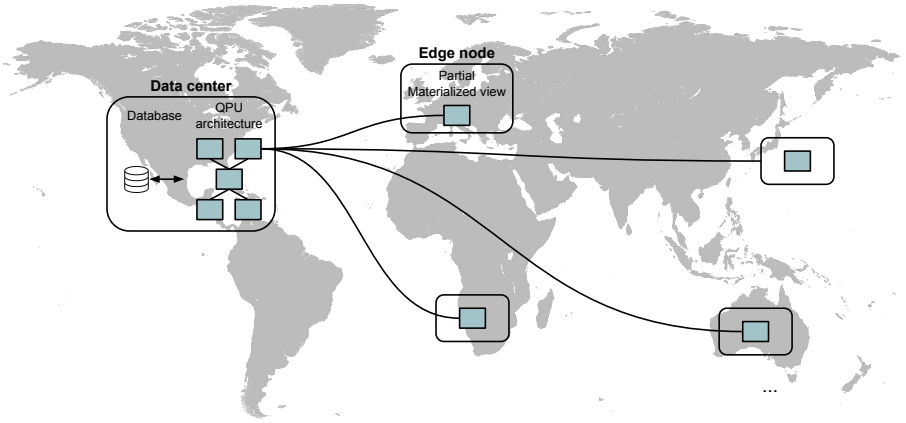


Figure 6.10: Distributing a QPU architecture between the data center and edge nodes.

6.3.2 Placing materialized views at the edge

Query response time is crucial for user-facing read-heavy application, such as Lobsters. As discussed in Chapter 3, even small increases in user-perceived latency can result in significant drops in both web traffic and sales. So far in this case study, we have examined how query response time can be decreased using pre-computed query results. Another factor contributing to query response time is the communication latency between the application and the clients. A client located in a different geographic region than the Lobsters application deployment might experience communication latency in the order of hundreds of milliseconds.

A common solution to this latency problem is to place on servers geographically closer to clients using caches, in order to avoid costly remote round-trips to data centers. These servers, called edge nodes, are a crucial component in industry architectures. For example, Google operates comparatively few data centers relative to edge nodes [6]. In existing applications, edge nodes are largely used for caching static data, such as images and video content, for example in content delivery network architectures.

Because of its modularity, the proposed query processing architecture can be used to place derived state, such as materialized views, closer to client. More specifically, the query architecture designed for the Lobsters application (Figure 6.9) can be distributed among the data centers and edge nodes. Considering a system architecture composed of a data center and multiple edge nodes, we can place a TopK materialized view QPU on each edge node, as shown in Figure 6.10. Each TopK-MV QPU stored pre-computed state for the K stories with the most votes. It thus provides low latency access to the data required for loading the application's frontpage, while maintaining a bounded memory footprint. We connect each TopK-MV QPU to the complete materialized view QPU and the SUM QPU placed in the data center, In that way, TopK-MV QPUs can receive updates for updated vote counts, and also request the materialize new stories as the reach the application frontpage.

We configure the connections between each TopK-MV QPU and the QPU architecture placed on the data center to be asynchronous, as propagating each vote to the edge nodes synchronously adds a significant overhead to vote operations. This choice makes a trade-off between write latency and the freshness of materialized views. Propagating updates to materialized views asynchronously means that views are eventually consistent, and might therefore return stale results.

In Chapter 8, we evaluate the effect of placing materialized view closer to clients on query response time and and query result freshness.

Chapter 7

Proteus: Towards application-specific query processing systems

We have implemented the composable query processing architecture described in Chapter 5 in the form of a framework that facilitates the definition and deployment of QPU-based query processing systems. We call the framework Proteus. Proteus consists of:

- A collection of implementations of QPU classes (Section 7.2).
- A service discovery mechanism that simplifies the configuration of a QPU architecture by allowing the QPU graph to self-organize with only local configuration input to each QPU instance (Section 7.1).
- An architecture description language that can be used to define the topology and configuration of QPU architectures (Section 7.3).
- A mechanism for translating architecture descriptions to deployment plans (Section 7.4).

In Section 7.5, we present the implementation details of Proteus.

7.1 Query processing domain dissemination

The set of queries that a query processing unit can process constitutes its query processing *domain*. As presented in Section 5.3.3.2, we encode the query processing domain using a tree data structure, called domain tree.

The domain of a query processing unit depends on its functionality (class), its configuration, and, in most cases, on the domains of its downstream connections as well. This is because QPU classes such as Join and Partition Manager process a given query by breaking it down to sub-queries, sending these sub-queries to their downstream connections, and then performing a computation over the returned results. Essentially, a QPU of this type de-composes queries into simpler tasks, and delegates some of these tasks to their downstream connections. Therefore, the set of queries that it can process depends on the types of tasks that its downstream connections can perform.

Because of that, a query processing unit requires the domain tree of each of its downstream connections for its operation. More specifically, as presented in Section 5.3.3.2, a QPU uses the downstream domain trees in order to: (1) compute its own domain tree, and (2) generate downstream queries during query processing. In this section, we describe the mechanism through which a query processing unit acquires the domain trees of its downstream connections.

7.1.1 Domain interface

We extend the query processing unit specification with a mechanism that allows a QPU to request the domain trees of its connections. For that, we define an additional interface, as follows:

$$GetDomain() \rightarrow [(QPUClass, DomainTree)]$$

An invocation of $GetDomain$ returns a stream that contains $DomainTree$ records, where $DomainTree$ is a serialization of a QPU's domain tree. When a QPU receives an invocation of its domain interface, it establishes an output stream with the caller. It then sends a $DomainTree$ record through the stream,

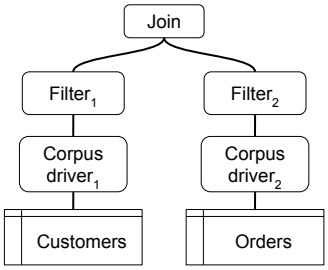


Figure 7.1: An example QPU architecture.

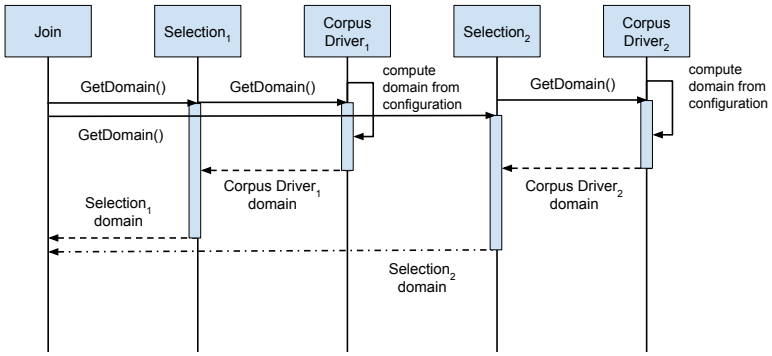


Figure 7.2: Sequence diagram for the domain discovery of the architecture in Figure 7.2.

representing its current domain tree. Each time there is a change to the QPU's domain tree, it sends an additional *DomainTree* record through the stream, representing the updated domain tree.

Similarly to the query interface, *GetDomain* requests follow the QPU graph topology: An edge in the QPU graph from QPU_A to QPU_B indicates that QPU_A can invoke the domain interface of QPU_B.

We define the domain interface response as a stream rather than an one-time response in order to enable propagation of configuration, domain and topology changes through the QPU graph. This is a first step towards enabling *dynamic reconfiguration* of QPU architectures. Examples of dynamic reconfiguration might include the creation new indexes or materialization of views at runtime. We consider dynamic reconfiguration of QPU architecture out of the scope of this work. It is, however, a direction for future work.

In order to support the domain interface, we extend the query processing unit specification with two additional methods: a domain processor and a domain response processor. These methods are analogous to the query interface methods. The domain processor method is responsible for processing a GetDomain request, and the domain response processor method is executed for each received domain response, and is responsible for updating the QPU's local view accordingly.

7.1.2 Query processing domain discovery mechanism

The first task of the constructor of each query processing unit class is to initialize the QPU's domain tree. For the Corpus Driver class this is performed using the input configuration (§5.3.3.2). Every other QPU class, as discussed above, requires the domain trees of its downstream connections in order to compute its own domain tree. Because of that, the constructor method of every QPU class except of the Corpus Driver starts by sending a *GetDomain* request to each of its downstream connections. After receiving the corresponding responses, the constructor computes the QPU's domain tree, using the rules presented in Section 5.3.3.2.

If a QPU receives a *GetDomain* request while still this process is being performed, it creates the response stream, but sends the response only after it has finished computing its domain tree.

We illustrate this process using the architecture shown in Figure 7.1 as an example. Figure 7.2 depicts a sequence diagram that describes the message exchanges for domain discovery in that QPU graph. Upon initialization, the *Join*, $Selection_1$ and $Selection_2$ send a *GetDomain* request to each of their downstream connections, $Corpus\ Driver_1$ and $Corpus\ Driver_2$ compute their query processing domains using their input configuration. Once each *Corpus Driver* QPU computes its domain tree, it responds to the corresponding *GetDomain* request. When each *Selection* QPU receives the *GetDomain* response, it calculates its domain tree, and then it responds to the request from *Join*. Finally, *Join* receives the responses from $Selection_1$, $Selection_2$, and computes its domain tree.

The goal of the domain dissemination mechanism is to reduce the input configuration required by QPU instances, in order to simplify the process of configuring and deploying a QPU architecture. Thanks to this mechanism, the configuration of a query processing unit consists of "local" configuration (configuration about that specific unit), as well as the endpoints of its downstream connections. The QPU can then use the domain interface of its connections to obtain information about them. For example, a Partition Manager connected to a number of index QPU instances requires only the endpoints of its connections as configuration.

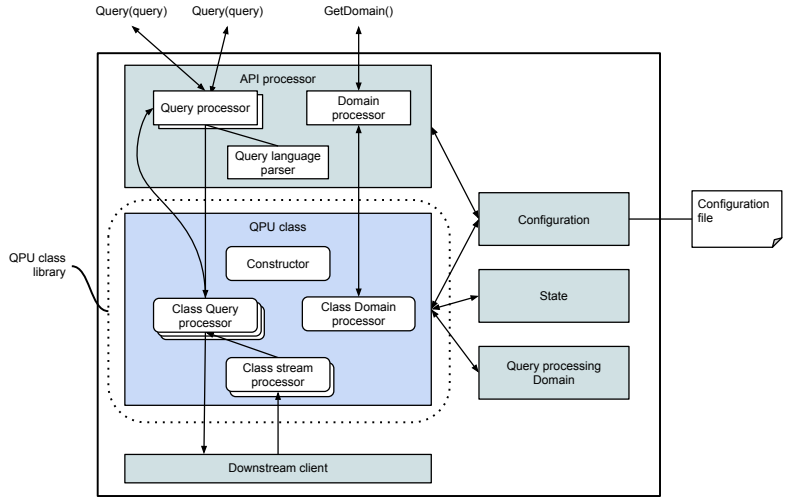


Figure 7.3: An overview of the QPU service architecture.

7.2 Query processing unit service

Proteus provides an implementation of the query processing architecture presented in Chapter 5. We have implemented the query processing unit component as a service, i.e. a daemon process. In this section we describe the system design and architecture of this service.

The design of the QPU service is guided by the principles of the query processing unit model. Instead of implementing a separate QPU service for each QPU class, we design the query processing unit service as a *polymorphic* service: Different instances of the same service implement different QPU classes, based on their configuration. To achieve that, we separate the components that are common to every QPU class, such as the configuration and query processing domain component, and the components that are class-specific. For the class-specific components, we provide implementations for different classes in the form a library. We ensure that components are separated and communicate through well-defined APIs.

Our goal with this design is to make the query processing service *extensible*. Implementing an additional QPU class consists of extending the QPU class library with component implementations for the additional class.

7.2.1 QPU service architecture

Figure 7.3 depicts the components of the query processing unit service in Proteus.

A configuration file is passed to the QPU service during initialization. The **Configuration** component is responsible for parsing the configuration file into a set of configuration parameters. It exposes an API that other components can use to retrieve these configuration parameters. Configuration parameters include the QPU class to be provided by the service, the endpoints of downstream connections, and class-specific configuration parameters. Some examples of class-specific configuration parameters are:

- The Corpus Driver class configuration specifies endpoints of storage tier components that the Corpus Driver communicates with to provide its functionality, the table it is responsible for, and, optionally, the table's schema.
- The Index class configuration, as described in Section 6.1, specifies an attribute name and an interval of values for that attribute.
- The Aggregator class configuration specifies an aggregation function, and the aggregation and grouping attributes.

The **Query Processing Domain** component is responsible for storing and providing access to QPU's domain tree as well as the domain tree of downstream connections. The QPU's domain tree is initialized by the QPU class Constructor method. The domain trees of the QPU's downstream connections are initialized and subsequently updated by the QPU class Domain Response Processor. The Query Processing Domain component provides two interfaces:

- An interface for providing a serialization of the domain tree to the QPU class Domain processor, and
- An interface for computing downstream queries based on a given query parse tree.

The **State** component is responsible for the QPU service's internal state. It provides a key-value interface which other components use for storing and retrieving state entries. The State component is used by derived state QPU classes, such as the Index and Materialized View classes, for storing their derived state structures.

We have defined a key-value interface for the State component, and have implemented multiple versions of the component using different backend stores: an in-memory implementation using an ordered map data structure, an implementation that uses MariaDB [124] as backend store, and one that use AntidoteDB [50]. The State component to be used by a QPU service instance is controlled by a configuration parameter.

The **API Processor** is responsible for receiving and processing incoming requests for the two open interfaces of the query processing unit: the Query and the Domain interface. It provides a Query Processor method which is responsible for processing query requests, and a Domain Processor method, responsible for domain requests.

When the Query Processor method is executed for query request received by the API processor, it first parses the given query to a query parse tree 5.3.3.1 using the Query Language Parser. If the query parse tree is successfully created, the Query Processor initiates a response stream, and invokes the Class Query Processor of the QPU class specified in the configuration. After executing the query, the Class Query Processor sends query result records back to the API Processor, which emits them at the response stream.

Similarly, when the API Processor receives a GetDomain request, it executes the Domain Processor, which passes the request to instance of the Class Domain Processor, and emits each received response as a stream record.

The API Processor can process multiple requests concurrently. For each received request, it creates an output stream, and spawns the corresponding Processor method in a new thread.

The **QPU Class** component is responsible for providing the methods defined by the query processing unit specification (Sections 5.2.1 and 7.1). It is implemented as a library that provides QPU class method implementations. Each class in the library provides five method definitions: a Constructor method, a Class Query Processor method, a Domain Processor method, a Query Response Processor method, and a Domain Response Processor method. In our prototype, we have implemented the following QPU classes:

- **Selection, Aggregation, Secondary Index, Partition Manager, Join, and Materialized view** as defined in previous chapters.

  As an optimization, we have integrated the functionality of the Selection class to other classes, such as the Corpus Driver, Secondary Index and Cache. This is because the Selections QPUs are often connected to these classes in QPU architectures. By integrating the selection functionality with these classes, we simplify QPU architectures and reduce the overhead of message serialization and de-serialization.

  Furthermore, in our current implementation, a Secondary index QPU service is responsible for a single attribute in a single continuous value interval, specified by its configuration. A multi-attribute query is decomposed into a conjunction of multiple single attribute queries.

- **Cache**. We have implemented a cache QPU class that stores query responses in an in-memory ordered map data structure. It uses a Least Recently Used eviction policy. Moreover, we defined the cache size in terms of *query result records*. That is, a query result that contains N data items occupies N units of cache size.

  Moreover, we have implemented a time-based and an notification-based invalidation policy. In the time-based policy, cache entries are invalidated after an amount of time specified by the QPU's configuration. In the notification-based policy, the cache QPU performs, for each query, a downstream persistent query to subscribe to notifications for updates that modify the query result; When a query result changes, the cache QPU invalidates the corresponding cache entry. The invalidation policy is controlled by a configuration parameter.

- **Aggregator.** We have implemented a generic Aggregator Class that divides an input set of data items into groups based on a grouping attribute, and applies a function over an aggregation attribute to the data item of each group. The grouping and aggregation attributes as well as the aggregation function are configuration parameters. An aggregation operation fails if any data item in the input does not have a value for the grouping or the aggregation attribute. Our current implementation provides the sum and count functions.

- **Corpus Driver.** We have implemented three versions of the Corpus Driver class, corresponding to different storage systems.: a relational database (MySQL [102]), a key-value store (AntidoteDB [50]), and an object storage system (Scality's CloudServer [114]). We present more details on the implementation of each Corpus Driver in Section 7.5.

- **Network.** We have defined and implemented an additional class, called Network. A Network QPU has a single downstream connection. It forwards every received query request to that connection,

and then forwards the input stream as its output stream. Moreover, it can be configured to delay, re-order or drop stream records, based on a specified distribution. The goal of this class is to simulate various network conditions for testing purposes.

The QPU class library is extensible; Additional classes can be implemented by providing the corresponding method definitions.

The **Downstream client** component is responsible for sending downstream requests to other QPU services. It exposes an interface other components can use to submit query and domain requests. The Downstream clients sends these requests to the corresponding connections, and for each received response records it invokes the corresponding Class Response Processor method.

Moreover, the Downstream client component is responsible for delivering the records of a each stream exactly once and in-order to the QPU Class. To achieve this, the API processor assigns a sequence number to each emitted stream record. The Downstream client uses these stream numbers to determine if records have been lost, already received or re-ordered. Moreover, the Downstream client can request from downstream connections to re-transmit records that it has missed, using control messages in the stream. The API Processor in turn stores stream records after sending them in a buffer of configurable length.

7.3 Architecture specification language

In this section, we present a simple architecture description language for describing QPU-based query processing architectures.

The goal of a an architecture description is to describe a QPU graph, including the class and configuration of the QPUs at its vertices and the connections among them, as well as the placement of the graph vertices across system nodes.

An architecture description comprises a series of placement context assignments, a series of QPU instance declarations, and a series of declarations of connections between QPU instances.

A placement context is denotes a node or collection of nodes. It has the following syntax:

```
PlacementCtx([endpoint], PlacementCtxName)
```

Listing 7.1: Syntax used for placement context assignment.

where [*endpoint*] is a collection of system nodes, expressed as hostnames, or IP addresses, and $PlacementCtxName$ is the name of the context assigned to these nodes. For example, the description:

```
PlacementCtx([10.200.4.56], "node_1")
PlacementCtx([10.200.3.45], "node_2")
PlacementCtx([10.200.4.56, 10.200.3.45], "dc_eu")
```

Listing 7.2: Assigning placement contexts to four nodes.

assigns the node with address 10.200.4.56 to the context "node_1" , the the node with address 10.200.3.45 to context "node_2", and both nodes to the context "dc_eu".

A QPU context declaration has the following syntax:

```
Configuration = {
  config_parameter_1: value_1,
  config_parameter_2: value_2,
  ...
}

qpu_object = <QPU_class>(Configuration)
```

Listing 7.3: Syntax used for specifying the configuration of a query processing unit.

where $< QPU_class >$ denotes one of the classes available in the QPU library. The configuration specifies the configuration parameters passed to the QPU during initialization.

Specifying a connection between QPUs has the following syntax:

```
qpu_object_1.connectTo(qpu_object_2)
```

Listing 7.4: Syntax used for specifying a connection between two query processing units.

This defines that *qpu_object_1* has a downstream connection to *qpu_object_2*.

Finally, the following syntax can be used to describe the placement of QPUs across system nodes:

```
qpu_object.place([PlacementCtxName])
```

Listing 7.5: Syntax used for specifying the placement of a query processing unit

The *place* method assigns a set of *placement constraint* to query processing unit. The QPU can be placed on any node that satisfies the constraint. For example, based on the description:

```
PlacementCtx([10.200.4.56], "node_1")
PlacementCtx([10.200.4.56, 10.200.3.45], "dc_eu")

q1 = Selection(config_selection)
q2 = Cache(config_cache)

q1.place("node_1")
q2.place("dc_eu")
```

Listing 7.6: Creating two query processing units, and specyfing their placement on system nodes.

$q1$ is placed on the node with address 10.200.4.56, while $q2$ can be placed on either of the two nodes.

A limitation of this configuration language is that it does not explicitly express the placement of a QPU graph relative to the corpus and the client, but rather it expresses placement on system nodes.

7.4 Query processing system deployment

We have designed a Deployment Generator component in Proteus for translating QPU architectures defined using the configuration language presented in the previous section into deployment plans.

Each QPU service runs in a Docker [94] container. The Deployment Generator translates an architecture description into (1) a Docker Swarm stack file and (2) a configuration file for each QPU service. The architecture is then deployed using Docker Swarm [12]

Docker Swarm uses Compose files [11] as deployment specification files. A Compose file is a YAML [13] file defines a set of services, a service being defined as a set of replicas of container that share the same image. The Deployment Generator creates a Compose file that defines a service for each QPU instance defined in the architecture description. A configuration file is passed to each service, using a shared volume. Moreover, the Deployment Generator specifies a common network that all QPU services are connected to.

In order to generate the configuration file to be passed to each QPU service, the Deployment Generator translates the configuration parameters of each QPU instance in the architecture specification to a TOML [110] file. Moreover, it adds to the configuration parameters the QPU instance's downstream connections as defined in the architecture specification.

7.5 Implementation

The prototype implementation of Proteus consists of 13k lines of Go (v1.14). The source code is available at https://github.com/dvasilas/proteus.

Interface. Applications interface with Proteus through a protocol buffer [63] interface over gRPC (v1.31) [67] gRPC is an remote procedural call (RPC) framework. Using gPRC, a client process can directly call a method on a server process, on a different machine, as if it were a local object. The workflow for using gPRC consists of defining a service, specifying its interface; methods that can be called remotely and their parameters and return types. gRPC uses protocol buffers as its interface description language. We selected gRPC due to its support for streaming and bidirectional streaming RPCs, and because it provides compiler plugins that can generate client- and server-side code based on a service definition.

Each query processing unit includes a gRPC server, as well as a gRPC client for communicating with other QPUs. Client applications also need to implement gRPC clients.

We have additionally implemented two clients libraries, in Go and Java. In the Go driver library, the operation of opening a connection to QPU actually opens and maintains a pool of connections to that QPU. Performing a query operations consists of requesting an idle connection, potentially opening a new connection if no idle connections are available, or, if the maximum number of connections has been reached, waiting until a connection becomes available, performing the operation, and returning the connection to the pool.

Storage. Overall, query processing units maintain state either in memory, or persistently in AntidoteDB [50] or MariaDB (v.10.3.27) [124]. However, currently not all QPU implementations support all three state backends. For example, our index QPU implementation supports both in memory and AntidoteDB, while

the Aggregator and join QPU implementations support only MariaDB. For QPU implementations that support more that one backends, the backend by a QPU instance is specified by a configuration parameter.

Subscribe API (see models chapter for terminology). As described in Section 2.1.1.3, the data storage tier exposes a logical Subscribe API that can be used for subscribing to notifications for data items updates. We have implemented the Subscribe API in three data stores: MySQL/MariaDB, CloudServer, and AntidoteDB [116], a REST server that compatible with the AWS S3 API, implemented by Scality. As a results, Proteus currently can be used with a data storage tier provided by one of these data stores.

We note that MySQL/MariaDB and AntidoteDB can be used both as a QPU state backend and as a storage tier. These two roles are independent: A QPU graph may be used to provide materialized view on top of AntidoteDB, and independent instances of AntidoteDB also be used for maintaining the state of QPUs in the graph;

MySQL/MariaDB: We implement the Subscribe API using a user-defined function [126], executed in response to a trigger [125]. One the same machine as the MySQL instance, we run an executable, termed notification server, that 1) implements a gRPC server, and 2) listens for socket connections. For each MySQL table associated with Proteus, we install triggers that call a user-defined function for each modification to the table. For each record that is inserted or updated, the user-defined function encodes and sends a message to the notification server through a socket connection. corpus driver QPUs can subscribe to this update stream using the notification server's gRPC service. The notification server is schema-agnostic; however, making a table available to Proteus requires installing the corresponding triggers for that invoke the user-defined function.

CloudServer CloudServer does not provide trigger or user-defined function mechanisms analogous to MySQL. We have thus modified CloudServer adding a notification service provided by a gRPC server. Our implementation consists of 300 lines of Node.js. The source code is available at `https://github.com/dvasilas/cloudserver/tree/proto/proteus`.

AntidoteDB Similarly to CloudServer, we have modified AntidoteDB adding a simple notification service that can be used by corpus driver QPUs. The source code is available at `https://github.com/dvasilas/antidote/tree/proteus/log_propagation`.

Operator implementation. The implementation of the core QPU functionalities, including relational operators (e.g. join, aggregation) and query processing data structures (e.g. secondary indexes, materialized views) are fairly straightforward. This is because update operations work in an incremental fashion, processing records as they arrive.

In addition, we note that, despite the fact that the *Query*() API allows for queries with fine-grained timestamps. most QPU implementations currently support three more coarse query modes: snapshot queries, persistent (or subscribe queries), and a combination of the two. Snapshot queries are executed on the most recent snapshot, in a best-effort fashion, and submitting a query request for an earlier snapshot returns an error.

A notable exception is the index QPU, in which we have implemented a versioned secondary index. In the versioned secondary index, each update creates a new version of the corresponding posting list. The index maintains a configurable number of versions for each data item; old versions are deleted to reduce memory/storage resource consumption. The index QPU maintains its state either in memory or in AntidoteDB.

Multi-threading. Each QPU maintains two thread pools: one for executing query requests and one for processing input stream records. Query requests and processing of input records are submitted as tasks to be executed by available threads. Our benchmarking showed that using worker pools results in improved performance and stability, especially in the implementations with MySQL backend, We attribute this to having a bounded number of threads performing database transactions, since our previous implementation was creating a new thread for each incoming request.

Packaging and deployment.

As mentioned above, the query processing unit service is packaged as a Docker container. This container is common for all QPU classes: the QPU's class configuration is specified by a configuration file passed to the service at initialization, Deploying a QPU service consists of deploying the QPU container and providing the configuration file as a shared volume.

We use Docker Swarm for deploying a QPU graph: A Docker compose file specifies the QPU services to be deployed, the configuration file to be passed to each service, and the placement of each service.

Chapter 8

Evaluation

In this chapter, we aim to validate our analysis of the trade-offs involved in query processing state placement decisions, and to evaluate the effectiveness of our design and prototype implementation in providing an effective mechanism to applications for navigating these trade-offs.

This evaluation consists of two parts. The first part (§8.1) is based on the case study of a read-heavy web application, presented in Section 6.3. It focused on the placement of a materialized view, and examines two placement options: in the data center, close to the corpus, or at the edge close to the clients. The aim of the first part is to answer the following questions:

- What is the overhead of materializing state in Proteus, directly over the data storage tier? (Section 8.1.3)
- What improvements in query processing performance can be achieved by placing materialized state close to clients? (Section 8.1.3)
- What are the penalties in freshness incurred when placing materialized state away from the data storage tier (Section 8.1.4)? How is freshness affected by load (§8.1.4.1) and round-trip time between sites (§8.1.4.2)?
- What additional data transfer costs result from materialized state placed close to clients receiving updates? (Section 8.1.5)

The second part (§8.2) of the evaluation is based on the case study of federated query processing on a multi-cloud corpus, presented in Section 6.2. In this part we examine three partitioning and placement configurations for a multi-cloud secondary index. We compare the three configurations under different types of workloads, and evaluate three metrics: query processing performance, freshness and data transfer costs.

The aim of the second part is to (1) demonstrate the expressiveness of the QPU approach, by deploying and comparing three query engine configurations across 3 data centers without any changes to the client application and the storage tier, and (2) validate that by controlling the configuration of QPU-based query engines, applications can navigate the trade-offs of geo-distributed query processing and optimize for different criteria according to their needs.

8.1 Placing materialized views at the edge

8.1.1 Experimental scenario

The evaluation in this section is based on the case study presented in Section 6.3, which describes the Lobsters [51] web application. We choose this application for the following reasons:

- It is characterized by a query-heavy workload that requires the materialization of derived state, and derived state is updated by a stream of small updates. This make Lobsters suitable for evaluating the efficacy of our design and prototype implementation in navigating the trade-offs of query processing state placement, by examining the effect of different placement schemes in query performance and query result freshness.
- It open-source [52], allowing us to examine the application's interaction with the database in which it stores its state, and statistics about the application's data usage patterns are available [104].
- It resembles a class of popular large-scale web applications, such as Reddit and Hacker News.

In Lobsters, users post, comment, and vote on "stories". Each story is associated with a "hotness" value that indicates how popular it is. Stories are ranked by hotness; the stories with the highest hotness value

appear on the front page. The hotness value of a story depends on parameters such as the number of votes for the story, the number of comments, and the hotness of those comments. Various operations, such as voting or commenting on a story, modify the hotness value. Computing the hotness value when it is queried would impose a prohibitive delay on queries In particular, serving the Lobsters' front page requires computing the hotness of every story in order to rank them. That is why the Lobsters application adds an additional column to the *stories* table which stores a pre-computed hotness value for each story. The application updates the value of the hotness column when operations are performed, such as upvoting or downvoting a story, or adding a comment to a story.

For this evaluation, we consider a version of the Lobsters application, that consists of two operations: voting for stories, and requesting the front page. We choose this simplified model of the application because it gives us better control over the properties of the workload for the purposes of this evaluation, while also capturing the aspects of the Lobsters application that make is suitable for this evaluation.

In particular, we consider the following database schema:

```
TABLE users (id bigint, username varchar(50))
TABLE stories (id bigint, user_id bigint, title varchar(150), description
    mediumtext, short_id varchar(6));
TABLE votes (id bigint, user_id bigint, story_id bigint, vote tinyint);
```

Listing 8.1: Simplified Lobsters schema used in this evaluation.

The front page is a listing of the 25 most highly ranked stories, including their title, author, and vote count. In the statistics provided by the Lobsters administrators, the front page operation constitutes 30.1% of client requests, and voting on stories constitutes 0.5% of client requests. The workload used for this evaluation consists of 95% front page operations, and 5% voting operations, unless otherwise specified.

Number of votes	[0-10)	[10-20)	[20-30)	[30-40)	[40-50)	[50-60)
% of stories	41.1	40.3	11.3	4.2	1.6	1.3

Table 8.1: Distribution of votes to stories in the Lobsters statistics [104].

According to the available data [104], most stories (81.4%) receive between 0 and 20 votes, while about 7% receive 30 votes or more. Our experiments show that when votes follow this distribution, very few votes are performed on the stories on the front page to have meaningful impact on query result freshness. To address that, we use a more skewed distribution: We configure 60% of votes to target the 25 stories in the front page, and 40% to follow the distribution shown in Table 8.1.

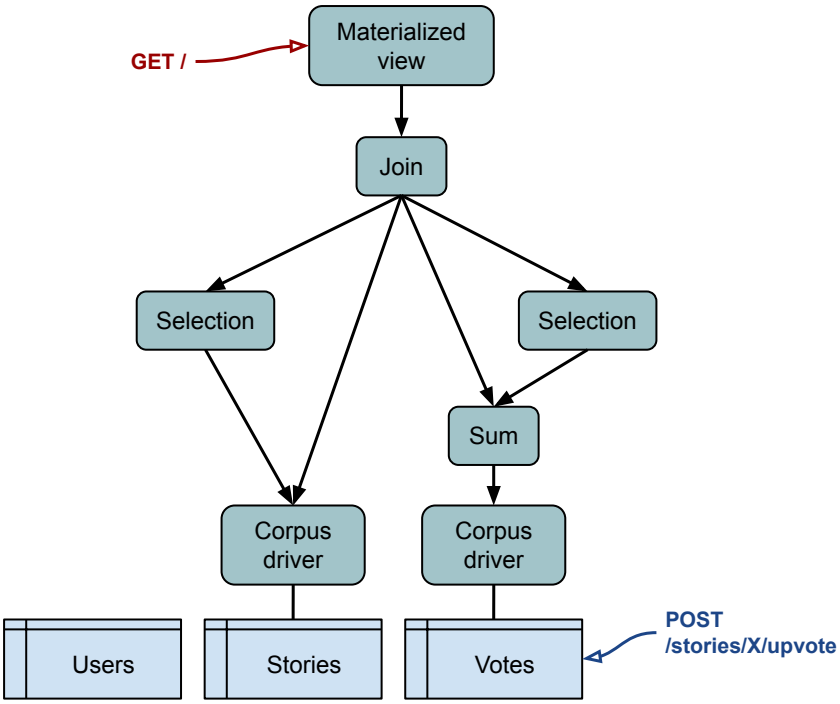


Figure 8.1: QPU graph used for this evaluation. The materialized view QPU computes the vote count for each story, and joins it with the corresponding record from the *stories* stable. The "GET /" request indicates the front page operation, and the "POST /stories/X/upvote" request indicates the operation of voting up story X.

The evaluation uses the QPU graph depicted in Figure 8.1 to maintain a materialized view that pre-computes the vote count of each count and joins it to the corresponding record of the *stories* table. The materialized view is defined by the following query:

```
SELECT id, author_id, title, url, vote_count
FROM stories
JOIN (
  SELECT story_id, SUM(vote) as vote_count
  FROM votes
  GROUP BY story_id
) view
ON stories.id = view.story_id
```

Listing 8.2: Definition of the materialized view maintained by the QPU graph shown in Figure 8.1.

This simplifies both the vote and front page operations compared to the baseline Lobsters implementation: the vote operation does not need to explicitly update the vote count of the given story, as this is performed by the QPU graph, and the front page operation can be served from the materialized view.

In the actual QPU graph deployed for the experiments, each selection QPU is merged with its downstream connection, in a single unit that performs both functionalities. In addition, the materialized view is merged with the join QPU.

The materialized view QPU implementation used in these experiments stores its state in a MariaDB instance. This is a separate instance from the one that is used the Lobsters database. It is deployed alongside and only accessible by the materialized view QPU. The choice of using the same database both as the baseline and for storing the materialized view is aimed at eliminating the effect of the database performance from the evaluation results, providing a comparison that isolates the effects of placement.

Finally, we create an index on the table that implements the materialized view in order to support efficient retrieval of the stories with the highest vote count.

We consider a system topology consisting of two geographically distant sites: The Lobsters application is deployed on one site, called server site, and clients are located on another site, called client site. Round-trip time between these two sites is 80 ms. This corresponds to a scenario in which the Lobsters web application is deployed in a data centers in North America, and clients are located in Europe [24], or vice versa.

Our experiments make use of the ability to place QPUs strategically in the system topology. We use two placement schemes: One in which the materialized view QPU is placed at the server site, and one in which it is placed at the client site. We evaluate the effects and trade-offs of materialized view placement by comparing these two placement schemes.

8.1.2 Experimental Setup

While the actual Lobsters Ruby-on-Rails application is open-source, we use a simplified version of it in order to isolate its interaction with the database. We implement an adapter that translates front page and vote requests to the queries that the real Lobsters application would issue, and issues those queries either to the Lobsters database (MariaDB), or MariaDB and Proteus, depending on the experiment configuration. This allows us to isolate the interaction between the application and the database, which is the focus of this work, and remove other tasks that the Lobsters application performs, which quickly become a bottleneck.

We implement this adapter as a server-side component: it is deployed along with the database on the server site, and plays the role of a simplified web server: It exposes a gRPC endpoint, similar to the QPU gRPC server, and clients issue operations to it as RPC requests. We choose this configuration because our initial experiments showed that issuing a large number of concurrent transactions to MariaDB under a 80 ms client-server round-trip time results in errors both in the database, and the Go MySQL library.

Workload generation. We implement a workload generator [140] that is responsible for issuing requests to the Lobsters adapter. The workload generator uses an open-loop model [122]: it creates requests based on a target load (requests submitted per second) value; each request is executed by a separate thread (creating and destroying threads are low-cost operations because threads are implemented as Goroutines).

In addition, the workload generator measures throughput and response time. We define response time as the delay that a client application experiences between issuing a request and receiving the corresponding response. To capture the variance of response time, we use a histogram data structure provided by the Go implementation of gRPC [68] that accumulates values in a histogram with exponentially increasing bucket sizes, in order to compute response time percentiles.

Our experiments show that while the open-loop design is effective at generating the target load both under low and high client-server round-trip times (in closed-loop designs, high round-trip times result in lower offered load), it results in a positive feedback loop effect above a certain load threshold: When the system starts not being able to handle the offered load, response time increases. However, the workload generator keeps generating requests, and, because of the increased response time, more requests are ongoing concurrently. This puts more load in the gRPC server, increasing the response time even more. Even a small initial increase in response time triggers this feedback loop, which eventually increases response time more and more during the duration of the benchmark. To address this problem, we extended the workload generator with a mechanism that limits the overall number of requests (and thus threads) that can be ongoing at a given point in time, to a configuration-specified bound. When the bound is reached, additional requests need to wait for ongoing requests to be completed.

The concurrency bound mechanism is effective in avoiding the positive feedback loop effect. However, it also means that when running a benchmark for a given target load we need to specify a bound in the number of concurrent threads that is sufficient for reaching the target load. If the bound is too low, then the workload generator cannot generate enough load, but and but response time does not increase because the delay caused by waiting for available threads is outside of the response time measurement.

Freshness. The materialized state maintained by the materialized view QPU updated asynchronously. As a result, queries served from the materialized view might reflect state that is stale relative to the database state. The staleness of the materialized state is impacted by its placement: Placing the materialized view QPU at the client side entails a minimum 40 ms communication latency to the materialized view (80 ms round-trip time between sites). Throughout the rest of this chapter, we consider the terms freshness and staleness equivalent and use them interchangeably.

One of the aims of this evaluation is to quantify how stale query results become. To achieve that, we measure the staleness of the results returned by the front page operation, using the following metrics:

- **Update latency:** The delay between a vote being committed in the database, and the materialized view being updated with the new vote count.
- **Returned version:** The difference, in number of versions, between the version of a story record returned by a query, and the version that would have been read by querying the Lobsters database instead of the materialized view.

We collect these metrics as follows. For each vote operation, the Lobsters database logs the timestamp at which the corresponding transaction commits; the database then includes the commit timestamp the update record that it publishes to the QPU graph. When the update record reaches the materialized view QPU, the QPU stores the commit timestamp in an “update log” table. Moreover, the materialized view

QPU logs the timestamp of each view update in the update log, and the timestamp at the start of each query, in a "query log".

At the end of a benchmark run, the materialized view QPU performs a post mortem analysis: The update latency for each vote is computed by subtracting the update timestamp from the commit timestamp. The returned version for each front page story is computed by comparing the update and query logs.

A limitation of this mechanism is that it requires comparing timestamps taken on different servers. To address that, in the benchmarks in which we take freshness measurements, we deploy all system components (Lobsters database, QPU graph, workload generator) as containers, on a singly physical machine. Because they share a single OS kernel, all timestamps used for computing freshness metrics are based on the host operating system's clock, and thus can be meaningfully compared. We use the Linux tc utility [77] to simulate the 80 ms round-trip time between sites, despite all containers being deployed on a single server.

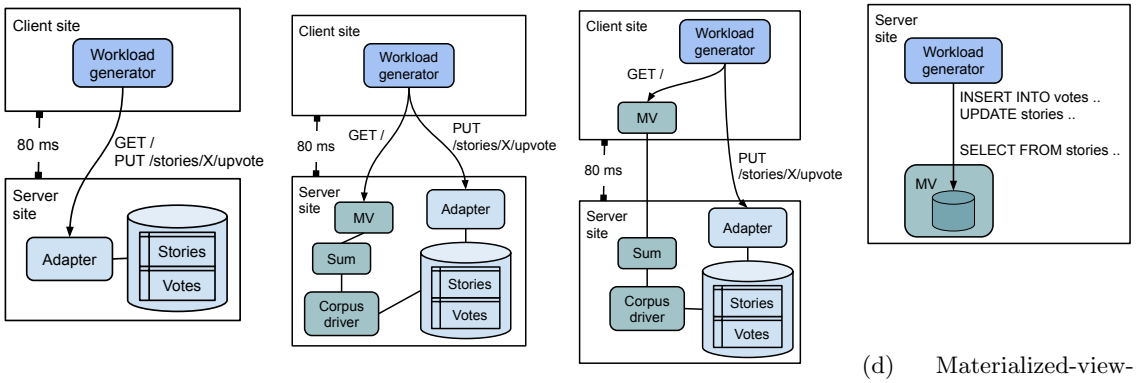
Hardware. Experiments were run on a cluster provided by the Laboratoire d'Informatique de Paris 6 (LIP6). Each server consists of 2 Intel Xeon E5645 CPUs, each with 6 cores, 64 GB RAM, an 128 GB SSD disk, and a 4 TB HDD disk.

Configuration. The average ping latency between machines in the cluster is less than 1ms. We simulate the two geographically distant sites by using the Linux tc utility [77] to add delay to outgoing packets.

For response time measurements, the Lobsters MariaDB instance and the QPUs, except the materialized view QPU are deployed on a single server on the server site, and the workload generator is on a server in the client site. The materialized view QPU is deployed on a separate dedicated server, either on the application or the client site, according on the placement scheme being tested. As described above, for the freshness measurements, all components are deployed on a single server.

Experiments run for 5 minutes unless otherwise specified, and we start taking measurements after an initial "warmup period" of 30 seconds. Repeated runs have shown that results are stable and consistent across runs.

8.1.3 Query processing performance



(a) Baseline/remote (b) Proteus/remote (c) Proteus/local (d) Materialized-view-internal

Figure 8.2: Deployments used in this evaluation.

We compare three deployments that differ in how they store and calculate the per-story vote count, and how they distribute computations and state across the nodes of the system (Figure 8.2) **Baseline/remote** (Fig. 8.2a) is equivalent to the real Lobsters application: it pre-computes and stores vote counts in a column of the Lobsters *stories* table. This serves as the baseline approach. **Proteus/remote** (Fig. 8.2b) consists of the QPU graph shown in Figure 8.1, deployed on the server site. In **Proteus/local** (Fig. 8.2c),the materialized view QPU is deployed on the client site. This is intended to evaluate the effect of placing the materialized view close the client. We also compare with a deployment (**Materialized-view-internal**, Fig 8.2d) in which the workload generator directly issues request to the materialized view QPU's state, bypassing the QPU's gRPC server (the workload generator and materialized view are co-located on the same server). This aims at providing an indication of the best performance the materialized view QPU can achieve, without taking into account its gRPC server (which we have identified as a performance bottleneck).

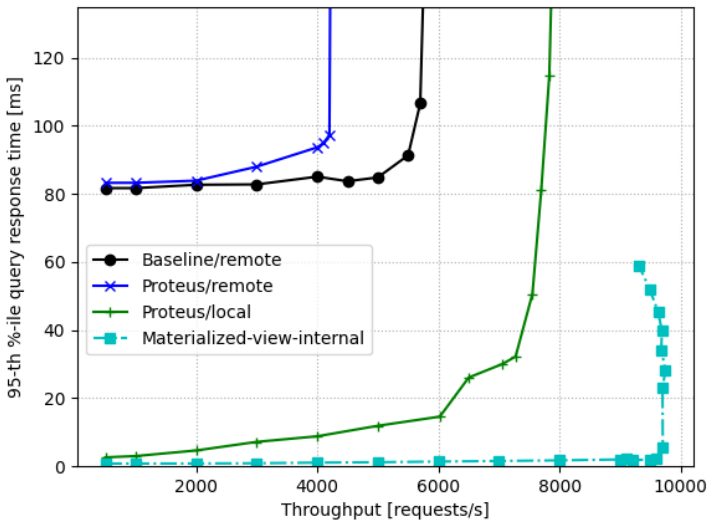


Figure 8.3: Throughput vs 95th percentile query response time.

Figure 8.3 shows throughput–query response time plots of these deployments. The ideal throughput–response time curve would be a horizontal line with low response time. The lower bound response time for Baseline/remote and Proteus/remote is 80 ms as this is the round-trip time between sites. In reality, all systems' plots have a "hockey stick" shape: latency remains relatively low until a point in which the system fails to keep up with the offered load. After that point, the system cannot achieve additional throughput, and response time increases.

Materialized-view-internal scales up to 9800 requests/second, outperforming both Proteus deployments. This deployment is intended to evaluate the performance of the core functionality of the materialized view QPU, by directly translating front page and vote operations to accesses to the QPU's state using a thread pool, bypassing the client-server gRPC communication. Because of that, we conclude that the gRPC communication incurs a significant overhead in the throughout that can be achieved by the system.

Proteus/remote scales to 4200 requests/second, which is a 24% overhead compared to the baseline deployment (Baseline/remote), which scales to 5500 requests/second. Both those deployments eventually read and write state to a MariaDB instance. The difference is how they translate requests to database accesses. The adapter used in the Baseline/remote deployment uses a simple logic that translates vote and front page request to database transactions. Conversely, the materialized view QPU used in the Proteus/remote deployment contains more complex logic, such as parsing received queries in SQL form, and receiving records from its input stream and updating the materialized view. We attribute the observed 24% overhead to the more complex logic.

In the Proteus/local deployment, query response time is significantly lower. This is achieved because moving the materialized view QPU to the client site removes the need for a costly round-trip to the server site, and thus removes the 80 ms lower bound. In addition, Proteus/local achieves a 28% increase in achieved throughput compared to the Baseline/remote deployment (we consider the maximum achieved throughput for which response time does not exceed 20ms and 100ms respectively). We attribute this improvement to the lower concurrency required to generate the same load in Proteus/local compared to Proteus/remote and Baseline/remote. In more detail, offering a certain load (volume of requests/second) requires creating a number of concurrent client threads, each performing a request. When the round-trip time between sites is 80 ms, each of these threads executes significantly longer compared to when the round-trip is just a few milliseconds. As a result, offering a given amount of load in the Proteus/remote setup results in a significantly greater number of threads, and thus open connections to the QPU's gRPC server, than in the Proteus/local setup. If the number of connections that can be opened is not bounded by a connection pool, this overloads the QPU's gRPC server, significantly increasing response times. When a connection pool is used, each requests needs to wait for an available connection, again increasing the end-to-end response time experienced by the client.

Conclusion. Our experiments confirm that placing materialized views closer to the client benefits read-heavy applications by removing costly round-trip communication across sites, and achieves scalability improvements. This results is expected, and shows the benefits that can be achieved by enabling this placement.

8.1.4 Freshness

8.1.4.1 Freshness vs Throughput

In the actual Lobsters application (and the Baseline/remote deployment), the vote count is maintained in the *stories* table. Because of the consistency guarantees of MariaDB, the vote count of a story, is always up-to-date with the state of the *votes* table. However, maintaining a materialized view placed at a remote site synchronously adds prohibitive overhead to write operations. Because of that the QPU graph in this evaluation maintains the materialized view asynchronously, and, as a result, its state might be stale relative to the state of the corpus.

In this section, we present, for the experiments describe in the previous section, the measurements for the freshness metrics presented in Section 8.1.2 (update latency and returned version). Our aim is to examine the effect of asynchronous derived state maintenance in the freshness of query results. We present results for both placement schemes (Proteus/local and Proteus/remote). Query results for the Baseline/remote deployment are always up-to-date, and update latency is 0.

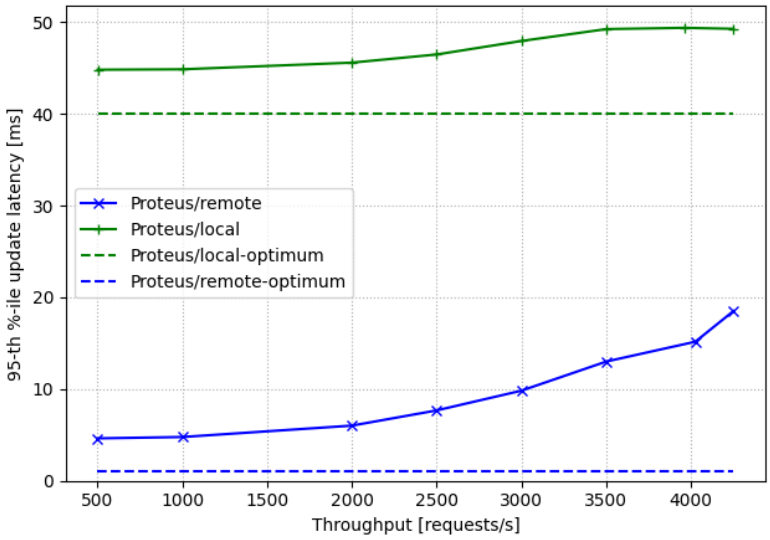


Figure 8.4: Throughput vs 95th percentile update latency.

Figure 8.4 shows 95th percentile update latency as throughput increases, for the Proteus/remote and Proteus/local deployments. As described above, update latency is the delay between committing a vote in the Lobsters database, and the corresponding vote count update in the materialized view. The ideal throughput–update latency curve would be a horizontal line with latency close to the lower bound defined by the communication latency. The lower bound for Proteus/remote is 40 ms while for Proteus/local it is less than 1ms. In reality, latency remains low as long as the system can keep up with the offered vote request load, and then increases.

Results show the Proteus/local setup scales well; Update latency remains within 5-10 ms of the lower bound. The Proteus/remote setup exhibits a higher update latency: For 4250 requests/second, the update latency in Proteus/remote is 88% higher than in Proteus/local, relative to the lower bound. This can be attributed to the same reasons as the scalability difference between the two deployments: The materialized view QPU in Proteus/remote is more loaded because more connections are concurrently ongoing for the same load value, compared to Proteus/local, leading to increased latency for applying updates to the view.

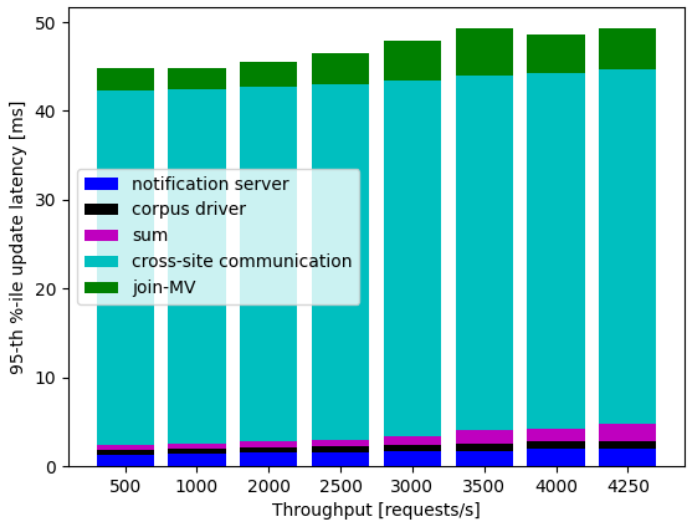


Figure 8.5: Breakdown of 95th percentile update latency in the Proteus/local deployment (as shown in Figure 8.4) as throughput increases.

Each vote committed in the database triggers a record that flows through the QPU graph, and eventually updates the corresponding vote count in the materialized view QPU. Figure 8.5 shows a breakdown of the update latency in the Proteus/local deployment. It depicts the delay at each step that vote records follow through the QPU graph.

A vote record's path consists of the following steps:

1. When the transaction that inserts a vote record into the Lobsters database commits, a trigger sends a message to the notification server (Section 7.5). The notification server then constructs an update record and sends it to the corpus driver QPU through a gRPC stream. The duration of this step is shown as **notification server** in Figure 8.5.
2. The Corpus driver receives an update record, and forwards it to the sum QPU (**corpus driver**).
3. The sum QPU receives an update record, computes an updated vote count, and sends a corresponding record to the join QPU (**sum**).
4. **Cross-site communication** in Figure 8.5 corresponds to the delay for sending an update record from the sum QPU, located at the server site, to the join QPU, located at the client site.
5. Finally, the join QPU receives a record and updates the materialized view accordingly (**join-MV**).

We observe that:

- Update latency is dominated by cross-site communication (up to 89%).
- Latency at the Corpus driver and sum QPUs is low: 2ms and 6ms at most. This is expected: the Corpus driver simply forwards records upstream; The sum QPU, for each record, updates a vote count stored in memory, and sends a record upstream.
- Latency at the notification server and materialized view QPU increases as the offered load increases. However, both scale well as the load increases. For the materialized view QPU this is due to the increasing load in the database that stores the materialized view. The latency increase in the notification server can be attributed to the increased load in the database, resulting in more triggers being executed concurrently.

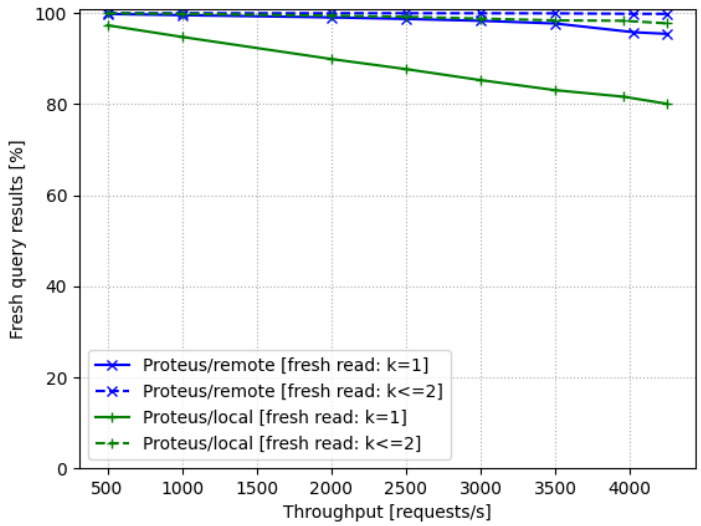


Figure 8.6: Throughput vs percentage of query results and that return a fresh version. A returned result is considered fresh if 1) it is the most up-to-date version committed in the database at that time (k=1), 2) it is amongst the two most up-to-date versions committed in the database at that time (k≤2).

Figure 8.6 shows the freshness of query results as throughput increases, measured as the percentage of query results that returned fresh versions. We define a returned result as a single story with its vote count; Each front page request returns 25 stories, and each is considered separately. We consider a scenario in which only the most latest version is considered fresh (k=1), and one in which the two most recent versions are considered fresh (k≤2).

We observe that:

- Proteus/remote has better freshness than Proteus/local, as expected. For the k=1 scenario, over 95% of reads observe the latest versions under the highest load. For the k≤2 scenario, freshness remains nearly constant at over 99%.
- Freshness in Proteus/local decreases as throughput increases. Proteus/local suffers from up to 80% stale query results (for k=1), and freshness decreases constantly as load increases. However, most stale results observe the second most up-to-date version: in the k≤2 scenario over 97% of query results are fresh.

These results can be explained using Figure 8.4. In Proteus/local, it takes at least 45 ms for an updated vote count to be reflected in the materialized view, but a front page request reaches the view with significantly lower delay. When load is low, this does not lead to stale query results because there are a few vote requests (5%). However, as load increases, queries observe increasingly more stale materialized view entries.

This is not the case for Proteus/remote. There, both types of requests reach the materialized view with similar delay, and because of the query-heavy nature of the workload, most queries observe tha latest materialized view entries.

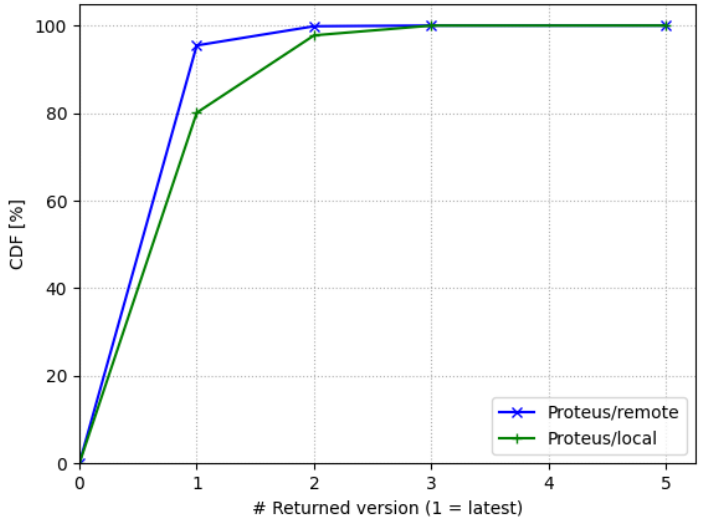


Figure 8.7: CDF of returned version at 4250 requests/second for the Proteus/remote and Proteus/local deployments.

Figure 8.7 displays a cumulative distribution function (CDF) showing how stale a front page request result is (measured in number of versions), under a load of 4250 requests/second, for the Proteus/remote and Proteus/local deployments. Results show that:

- In both deployments, most queries return the latest or second most recent version.
- Proteus/remote has better freshness than Proteus/local: Under the same conditions, Proteus/remote exhibits 4.5% stale returned results, while Proteus/local 20% (4.4X). However, in Proteus/local only 2.2% of queries results are more stale than the second most recent version.

Conclusion. Placing the materialized view close to the client, and thus away from the underlying datastore incurs a freshness penalty: queries return stale results relative to the results that would have been obtained by querying the database. However, for the workload characteristics in these experiments, query results are rarely more stale than the second most recent version. Moreover, update latency and versions freshness scale well as the system's load increases. We can argue that the level of freshness shown in these experiments to be achievable when placing materialized views close to the client is sufficient for many query-heavy applications that tolerate eventual consistency.

8.1.4.2 Freshness vs round-trip latency

The experiments in the previous section evaluated the effect of the query processing state placement under a constant round-trip delay, as the load offered to the system increases. In this section, we invert these two variables: we measure freshness under a constant load, as the (simulated) round-trip time between the application and client site increases, for the Proteus/local deployment. The aim of this experiment is to examine the effect of round-trip delay in freshness.

Experiments are performed under a load of 2000 and 4000 requests/second. We have selected these values based on the results shown in Figure 8.4: Under a load of 2000 requests/second both deployments are able to keep up with the offered load, while under 4000 requests/second the Proteus/remote scheme exhibits increased update latency.

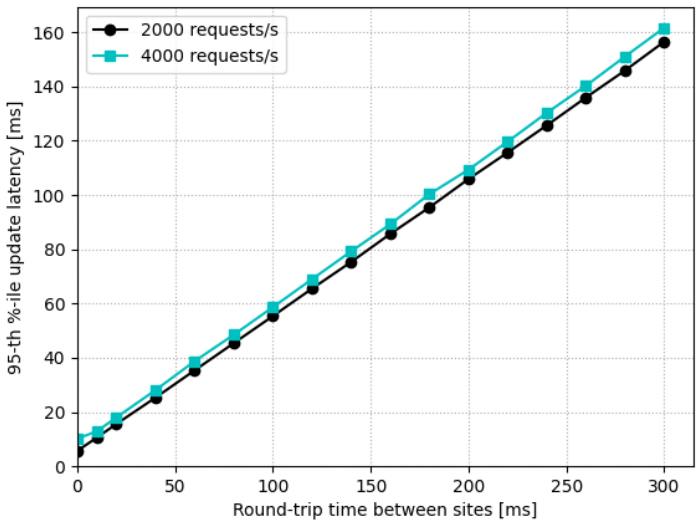
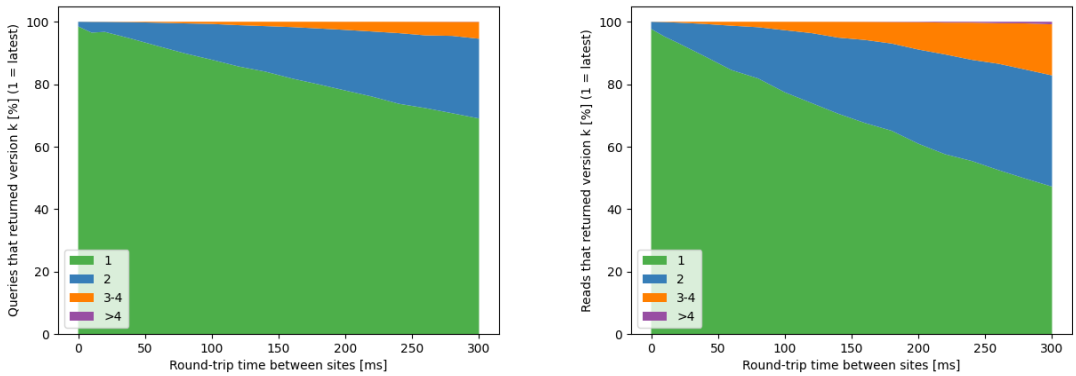


Figure 8.8: Round-trip time between sites vs 95th percentile update latency, for the Proteus/local deployment.

Figure 8.8 shows the update latency as the round-trip time between the two sites increases, under 2000 and 4000 requests/second. We observe that for both loads, update latency scales linearly with the round-trip time. Under 2000 requests/second, update latency is at most 5ms above the lower bound set by the one-way network latency between the two sites, while under 4000 requests/second it is at most 11ms (2.2X).



(a) 2000 requests/second.

(b) 4000 requests/second.

Figure 8.9: Distribution of returned version vs round-trip time between sites.

Figures 8.9a and 8.9b show the distribution of returned version (which version relative to the most recent one was returned by a query) for 2000 and 4000 requests/second respectively. We observe that under both loads freshness decreases as round-trip time increases; Increasing the load of the system increases the gradient of this decrease. However, in both cases, queries, generally, observe at most the forth most recent version: Only 0.06% and 0.8% of query results are more stale than the forth most recent version.

Conclusion. Update latency, and the freshness of query results are primarily affected by round-trip time between sites, and to a lesser degree by the system's load.

8.1.5 Data transfer between sites

Deployment	Baseline/remote	Proteus/remote	Proteus/local
Cross-site data transfer (MB)	0	0	7.7
Data transfer out to internet (MB)	≈7700	≈7700	≈7700

Table 8.2: Measured data transfer for a 5 minute benchmark with a load of 4000 requests/second.

Distributing a query engine across multiple sites entails data transfer between sites. If the system is deployed on a public cloud platform this incurs an additional cost because data transfer between data centers is part of public clouds' pricing models. For example, on AWS EC2, data transfer costs $0.02 per GB [2].

In the scenario used for these experiments, placing the materialized view QPU at the client site entails that update records from the Sum to the materialized view QPU are sent between sites.

To measure the amount of inter-site data transfer, we have implemented a mechanism for measuring and aggregating the size of outgoing messages at each QPU. For a 5 minute benchmark, with 4000 requests/second (200 votes/second), 7.7MB of data were transferred between data centers. This is because only 5% of requests are votes, and the size of an update record is small (around 90 bytes), as it only contains the id and vote count of a story.

In contrast, in the same benchmark, 7.5GB of data were sent as query responses. This is because the size of a query response is around 4MB (it contains the records of 25 stories), and 95% of requests are queries.

We conclude that, in this the evaluation scenario, the materialized view can be placed in the client site without incurring significant data transfer costs.

8.1.6 Conclusion

The evaluation presented in this section demonstrates that there are benefits and drawbacks to both placement options: Placing materialized views close to the client results in improvements in response time and throughput, at the expense of freshness. Conversely, placing materialized views close to the corpus ensures fresh query results, but brings limitations to response time and throughput. As a result, client-site placement of materialized views is more suitable for applications that require low query response times or high query load, and can tolerate stale query results; server-site placement is better-suited for applications for which query processing performance is not critical, but require up-to-date query results. In addition, evaluation results show that Proteus can to efficiently implement both placement schemes.

8.2 Federated metadata search for multi-cloud object storage

8.2.1 Experimental scenario

The evaluation presented in this section is based on the case study presented in Section 6.2. We consider a multi-cloud data serving system, composed of three storage locations. A storage location may be a public cloud storage platform (e.g. Amazon S3, Microsoft Azure Blob storage, Google Cloud Storage), or an on-premise storage system. Each storage location is independent (not aware of the other locations), and a multi-cloud data controller (§6.2) is responsible for providing a common namespace across storage locations. The controller implements an object storage API, such the AWS S3 API: objects are composed of a primary key, a set of metadata attributes, and content. This evaluation focuses on providing support for federated queries on metadata attributes.

We consider a system model composed of the 3 storage locations, in 3 geographically distant data centers. Each storage location stores a disjoin subset of the dataset. An instance of the multi-cloud controller is deployed on each storage location, and users are served by the controller that is geographically closer to their location. An overview of the system model is shown in Figure 8.10.

We consider an application that consists of two types of operations: updating the metadata attributes of a given object, and performing queries on metadata attributes.

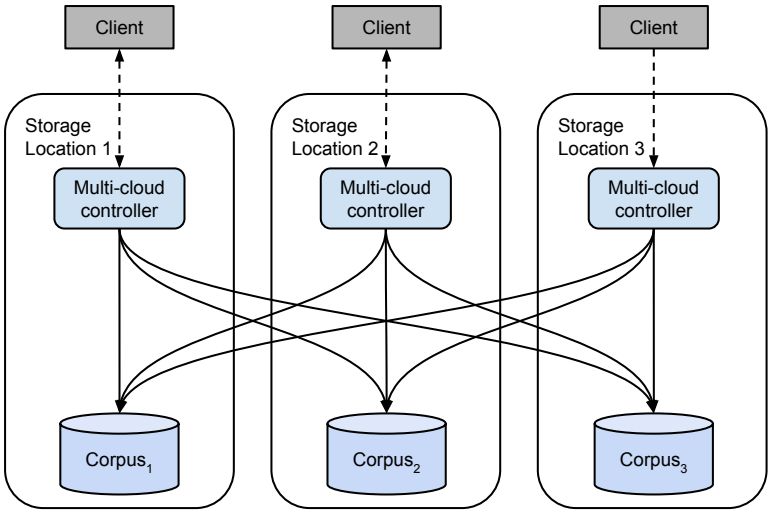


Figure 8.10: An overview of the system model in §8.2.

8.2.2 Methodology

As discussed in Section 6.2, there are alternative approaches for designing a multi-cloud query engine. The aim of this evaluation is to demonstrate the need for flexibility in the query engine's design for addressing the needs of different applications, and validate that QPU-based query engines can provide the required flexibility.

To achieve that, we consider 3 query engine configurations (QPU graphs), 3 workload types with different query-update ratios, and 3 metrics (query processing performance, freshness, and data transfer between storage locations); We experimentally determine which query engine configuration is better-suited for each combination of workload type and target metric.

Query engine configurations. The main functionality of the query engine in this scenario is to maintain secondary indexes for accelerating queries on metadata attributes. We consider the following approaches for partitioning and placing a multi-cloud secondary index across the system.

- **Replicated global indexes (rg-index):** An index responsible for indexing data from all storage locations is deployed on each location. This approach has the advantage that queries are served by the local index. However, each index needs to receive update notifications from the two remote storage locations. Depicted in Figure 8.11a.
- **Partitioned index (p-index):** In this configuration, the index on each storage location is responsible for the local corpus. The system forwards each query to all 3 storage locations, and combines the retrieved results. This approach requires a third of the storage space for indexes compared to rg-index, and ensures that update notifications are sent only to the local index, at the expense of requiring cross-site communication for serving queries. Depicted in Figure 8.11b.
- **Partitioned index with caching (p-index-cache):** This configuration is an extension to p-index that uses a caching layer with the aim of reducing access to remote indexes. For each index, a cache responsible for caching sub-query results from it is deployed on the two remote storage locations. Depicted in Figure 8.12.

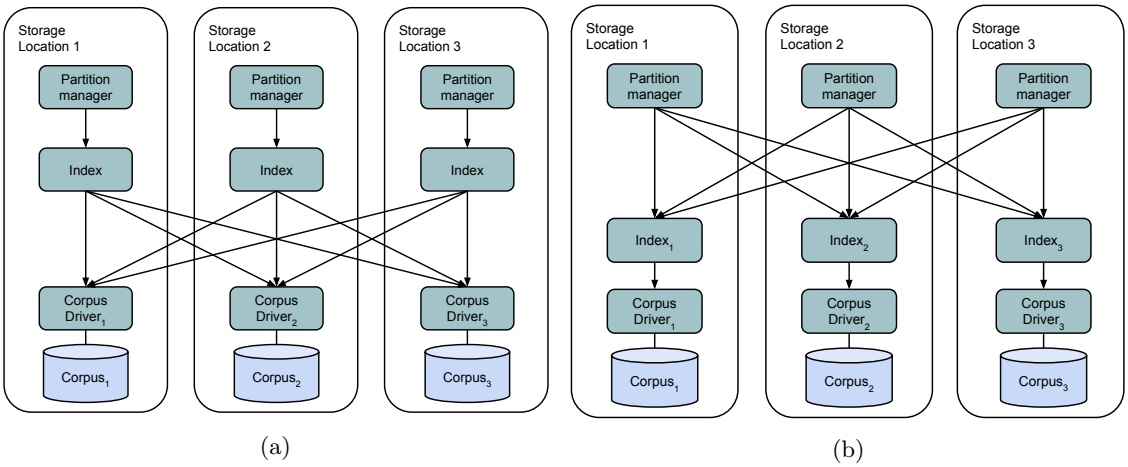


Figure 8.11: The (a) replicated global indexes (rg-index) and (b) partitioned index (p-index) QPU graph configurations.

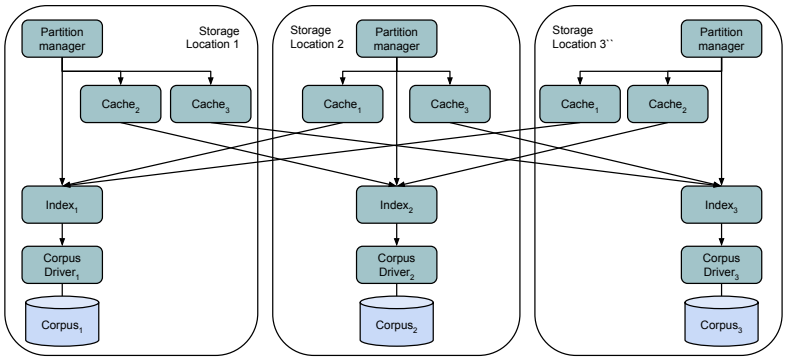


Figure 8.12: The partitioned index with caching (p-index-cache) QPU graph configuration.

Workload types. Overall, we consider a query-heavy application that requires the use of secondary indexes for query processing. We examine 2 workload types, each with a different mix of query and update operations: 95% queries - 5% updates (w95/5) and 50% queries - 50% updates (w50/50).

Evaluation metrics. For this evaluation, we use the metrics discussed in Section 8.1: query processing performance, freshness and data transfer cost for data transferred between storage locations.

8.2.3 Experimental Setup

On each storage location, data is stored on an instance of MongoDB. We use MongoDB as an object store: an object is represented by a MongoDB document, with document fields representing the object's metadata attributes. At the start of each experiment, we preload each storage location with with 33k objects, so that in total the system stores 100k objects.

The index QPU implements an in-memory B-tree secondary index. The cache QPU implements a cache with a least recently used (LRU) eviction policy, and a notification based invalidation policy. We set the size of caches so that each cache can hold 50% of the index entries it is responsible for.

We configure the system so that there is an 80sm round trip time between storage locations in order to simulate a multi-cloud system deployed over distant geographic locations.

Workload generation and measurements configuration. We use the Yahoo! Cloud Serving Benchmark (YCSB) [45] for generating workload and performing measurements. The core operation of the YCSB framework is that it drives a number of client threads, each executing a sequential series of operations by making calls to the underlying system, and measures the latency and achieved throughput of their operations. At the end of an experiment, a statistics model aggregates the measurements and reports the achieved throughput and the measured latency percentiles.

We have modified YCSB's MongoDB driver to send query operations to Proteus. We deploy a YCSB instance on each storage location. On each location, YCSB client threads send update operations to the local MongoDB instance, and query operations to the local partition manager QPU. At the end of an experiment, we gather and aggregate measurements from all three storage locations, and compute the total achieved throughput and latency percentiles.

Each object in the dataset has multiple, randomly generated metadata attributes. We ensure that a numeric attribute with a specified key existing in every object. Query operation in the workload are point queries that refer to this attribute. Both the attribute's values and query predicated follow a uniform distribution. We use the mechanism presented in Section 8.1.2 for measuring freshness.

Experiments run for 5 minutes unless otherwise specified, and we start taking measurements after an initial warmup period of 30 seconds.

8.2.4 Query processing performance

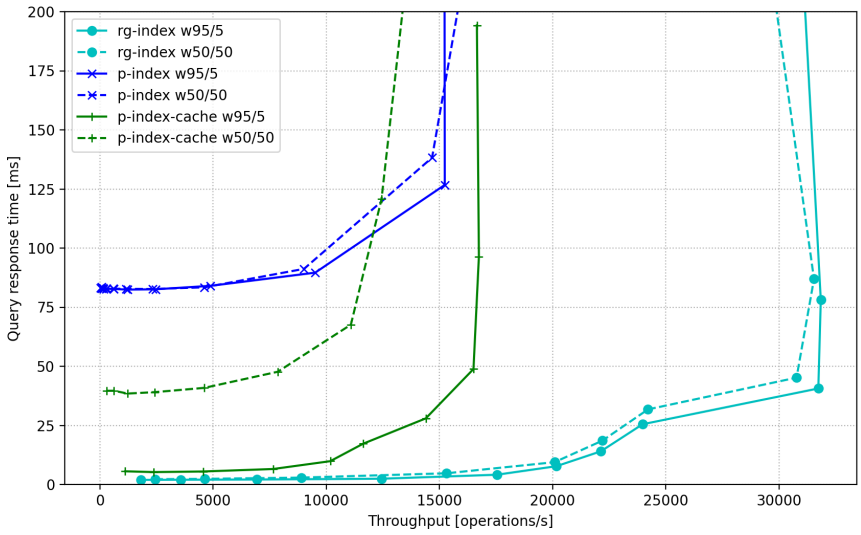


Figure 8.13: Throughput vs query response time. Plots for the rg-index and p-index configuration show the 90th percentile response time; Plots for the p-index-cache configuration show average response time in order to capture the effect of caching in response time.

Figure 8.13 shows throughput–query response time plots for the 3 QPU graph configuration and 3 workload types. We observe that:

- Response time in the p-index configuration is 80ms higher than in the rg-index configuration. This is because in p-index the query engine forwards queries to indexes across storage locations while in rg-index queries are served by the local index.
- The p-index configuration achieves around 50% throughput compared to rg-index. This can be attributed to the closed loop workload generation mechanism of YCSB: Given a certain number of client threads, in rg-index each query operation has a takes less than 25ms when the system is not saturated, while in p-index each query operation take 85-90ms because of the 80ms round trip time between storage locations. Therefore, in p-index each client thread cannot offer the same load as in rg-index.

We conclude that the rg-index configuration is better-suited for achieving the best query processing performance. However, this comes at the expense of memory overhead for maintaining a global index at each storage location. The rg-index-cache configuration requires achieve query processing performance comparable to rg-index, and requires 66% the memory of rg-index (because each cache is configured to 50% the size of an index).

8.2.5 Freshness

In this section, we examine the query result freshness achieved by the alternative QPU graph configurations. In order to evaluate the freshness of the different configurations, we break them down into *placement*

patterns, we evaluate the freshness of each placement pattern, and reason about how placement pattern freshness contributes to the overall freshness of QPU graph configurations. The placement patterns present in the three QPU graph configurations are shown in Figure 8.14.

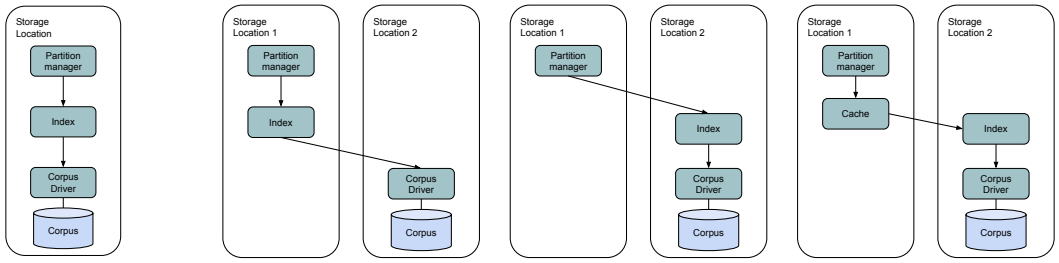


Figure 8.14: Placement patterns. (a) Corpus, index and clients are located on the same storage location (local). (b) The corpus is located on a remote storage location (remote-corpus). (c) The index is co-located with the corpus; clients are located on a remote storage location (remote-client). (d) A cache is used to reduce cross-location communication (remote-client-cache).

	local	remote-corpus	remote-client	remote-client-cache
rg-index	■	■		
p-index	□		■	
p-index-cache	□			■

Table 8.3: Placement patterns that QPU graph configurations are composed of. ■ ■ indicates that both patterns contribute to the configuration's freshness. □ ■ indicated that only the pattern with ■ contributed to the configuration's freshness.

Table 8.3 shows the placement patterns used for each QPU graph configuration. In rg-index, each index is connected to both local (8.14a) and remote (8.14b) corpus. As a result, query result freshness is affected by the freshness characteristics of both patterns. In p-index, each partition manager QPU is connected to the the local index (8.14a), and two remote indexes (8.14c). The Partition Manager forwards a given query to all three indexes and waits to receive all responses before responding to the client. Because of that, the freshness of p-index is determined by the freshness of the remote-client pattern. Similarly, the freshness of p-index-cache is determined by the freshness of the remote-client-cache pattern (8.14d).

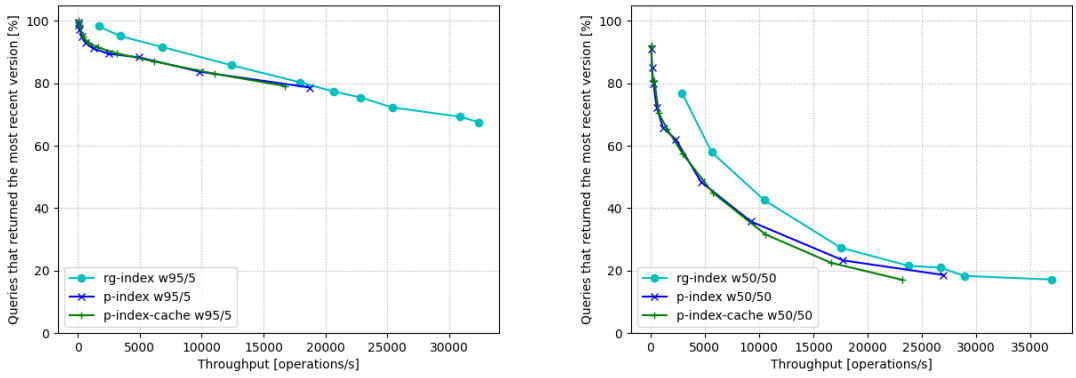


Figure 8.15: Throughput vs percentage of queries that returned the most recent version in the w95/5 (a) and w50/50 (b) workload.

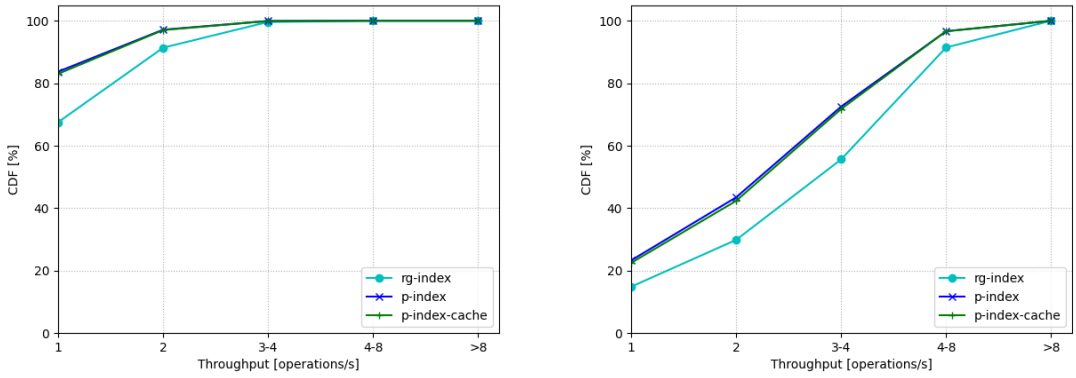
Figures 8.15a and 8.15b show the percentage of query operations that return the most recent version of an object as load increases, for the w95/5 and w50/50 workloads respectively. We observe that load has an impact on query results freshness, and this impact is greater when there are more update operations.

This is the result of the higher load of updates as well as lookup operations to the index, both of which need to acquire locks resulting in congestion.

In addition, we observer that the difference in freshness between p-index and p-index-cache is small. Indeed, we expect the cache to have a small effect to freshness. This is because in the case of a cache miss the freshness of the query is equivalent to the freshness of the remote index. In the case of a hit, reading a stale result requires that the cache lookup occurs in the interval between the remote index being updated and the corresponding notification reaching (and invalidating) the cache, which in this case is 40ms.

While Figures 8.15a and 8.15b demonstrate the impact of load to freshness, they are not meaningful for comparing the freshness of the three configurations. In order to compare the freshness of the three configurations, we select for each configuration the "nominal throughput" using Figure 8.13: The nominal throughput is the ideal throughput at which each system can run. We define it as the higher measured throughput value in which query response time is close to the response time that the system can achieve at a low load. These throughput values are:

- Rg-index: 22126 operations/second for w95/5 and 22196 operations/second for w50/50.
- P-index 9511 operations/second for w95/5 and 9015 operations/second operations/second for w50/50.
- P-index-cache 11633 operations/second for w95/5 and 7856 operations/second operations/second for w50/50.



(a) (b)

Figure 8.16: CDF of returned version in the w95/5 (a) and w50/50 (b) workload.

Figures Figure 8.16a and 8.16a show the CDFs of returned version for the measurements listed above. We observe that:

- In w95/5, queries in all configurations retrieve at least the second most recent version.
- In w50/50, queries retrieve up to 4-8 stale versions. This is because of the higher ratio of updates in the workload.
- In w50/50, rg-index retrieves more stale results than p-index and p-index-cache. This is expected due indexes receiving notifications from remote storage locations.

8.2.6 Data transfer between storage locations

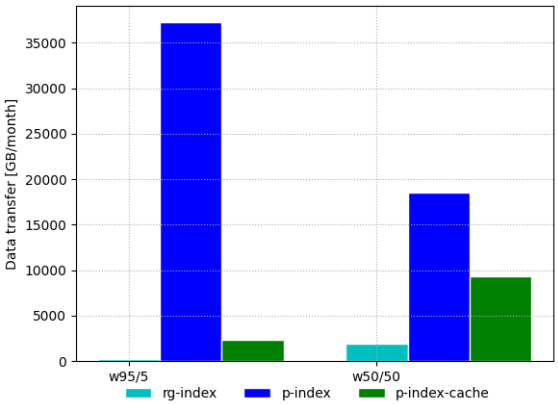


Figure 8.17: Monthly amount of data transfer between sites.

In this section, we examine the amount of data sent across storage location for the different QPU graph configurations and workload types.

In the rg-index configuration, cross-location data transfer occurs for sending update notifications from the corpus to remote index QPUs: Each update operation generates three update notifications, one for each storage location. As a result, data transfer increases as the ratio of updates increases.

In the p-index configuration, cross-location data transfer occurs for retrieving query results from remote index QPUs. partition manager QPUs forward every query to all three storage locations, and retrieve the results. As a result, data transfer increases as query ratio increases.

In the p-index-cache configuration, cross-location data transfer occurs on cache misses. When an index entry is not present in a cache, the cache QPU forwards it to the corresponding index QPU, and retrieves the results. As a result, data transfer increases as the ratio of cache misses increases.

We experimentally measure the amount of data transferred between storage locations for each workload - QPU graph configuration pair. We use the nominal throughput of each pair as the load offered to the system for each run. For each experiment, we measure the amount of data transfer between storage locations for the duration of the run (5 minutes), and use that measurement to extrapolate the monthly data transfer, as follows. From the amount of data transferred within 5 minutes we extrapolate the amount of data transferred in 2 hours by assuming that the average throughput for the 2 hours is that of the 5 minute benchmark. Then we extrapolate the data transfer in a day by assuming that those 2 hours are "peak hours", and contribute the 20% of the total daily data transfer. Figure 8.17 shows the results.

These results demonstrate another aspect of the trade-off between the rg-index and p-index configurations. P-index and p-index-cache use significantly higher data transfer than rg-index. This is because the size of an update notification (transferred in rg-index) is significantly smaller than the size of a query response (transferred in p-index and p-index-cache). For a given update, the update notification contains the object's primary key, the updated attribute(s) and a timestamp, while for a query, the response contains the primary keys, attributes and timestamps of all objects that match the query predicate.

Furthermore, we observe that using a caching layer results in data transfer reduction for the both workloads, as expected. This reduction is more significant for w95/5 as the lower ratio of updates results in fewer cache misses due to invalidations. Therefore, using the p-index-cache configuration can result in significant data transfer cost reduction compared to the p-index configuration.

8.2.7 Conclusion

Our evaluation results demonstrate that no single configuration is best-suited for all metrics: rg-index outperforms the other two in terms of query response time and provides the lower data transfer cost, while p-index and p-index-cache exhibit the best freshness when the workload contains a high number of updates.

However, applications are rarely interested in optimizing a single metric, but rather finding the right balance in the trade-offs between metrics. Our evaluation results indicate that, for query heavy workloads, the p-index-cache configuration offers a balance between query processing performance, freshness, and memory resource and data transfer overhead.

The evaluation presented in this section demonstrates that Proteus can be efficiently used to tune the trade-offs between query processing performance, freshness and operational costs by controlling the query engine's architecture.

8.3 Conclusions

The evaluation results presented in this chapter confirm our analysis of the trade-offs involved in the placement of derived state. This validates the central argument of this thesis: No single derived state partitioning and placement approach is optimal for all needs, and flexibility is required for addressing different needs.

Moreover, the evaluation demonstrates the expressiveness of the proposed approach: QPU-based query engines enable the flexibility for configuring derived state partitioning and placement with simple configuration changes to the query processing middleware, without requiring changes to the application. QPU-based query engines can be used to flexibly partition derived state, and place state and query processing computation freely across the system. THanks to that they can be employed to address diverse application characteristics and requirements.

Chapter 9

Related work

In this chapter, we discuss literature related to varying aspects of the work presented in this thesis.

The work presented in this thesis builds upon existing approaches in the areas of secondary indexing, materialized views, and caching. In Sections 9.1, 9.2 and 9.3 we outline design alternatives in each of these areas, and place our design existing approaches with respect to these. In more detail, in Section 9.1, we present secondary indexing systems proposed in the literature, focusing on index partitioning schemes. The main difference with these approaches, is that our design aims at flexibly supporting both partitioning schemes. Another goal of our design is to enable configurable derived state maintenance. In Section 9.1.1, we outline index maintenance strategies proposed in the literature, and present existing indexing systems that support multiple maintenance schemes. Our work expands on these designs by enabling configurable maintenance for materialized views in addition to indexes. In Section 9.2, we outline related work on systems that maintain materialized views for reducing query response time and increasing supported load. We present techniques that these works propose, and discuss how these techniques have motivated our design. Section 9.3 discusses query result caching. In particular, we present techniques for keeping cached results up to date as changes in the corpus occur, and discuss how these techniques can be implemented in QPU-based query engines.

Section 9.4 examines the problem of query processing when the corpus is distributed across multiple data centers. This problem has been studied in the context of multi-site web search engines. We present a taxonomy of multi-site search engine architectures, which have motivated the design of Proteus. In addition, we outline the differences between the problem multi-site web search and the problem of geo-distributed query processing studied in this thesis.

Two central properties of Proteus is the modular architecture, designed so that it can be assembled using basic building blocks, and its ability to enable configurable state and computation placement. Sections 9.5 and 9.6 discuss related works in these areas. In Section 9.5, we present approaches for modular and composable system architectures. While these works address different research problems (Click [79] addresses the problem of building configurable routers, and Pleiades [29] addresses the problem of enforcing large-scale distributed structural invariants), the techniques proposed by these works have motivated the modular architecture of Proteus. Section 9.6.1 and 9.6.2 discuss operator and placement techniques respectively.

Finally, the query processing unit computation model shares similarities with other computation models, including the MapReduce and dataflow computation models. In Section 9.7, we discuss the similarities and differences between the QPU computation model and these paradigms.

9.1 Secondary indexing in distributed data stores

A number of works [54, 76, 135, 136] have examined the challenges of implementing secondary indexing in distributed databases.

SLIK [76] discusses the design space for implementing a secondary indexing system in a distributed key-value store, including aspects such as index partitioning, consistency between corpus and index, and performing long-running operations such as index creation and migration.

SLIK uses the B+tree data structure for indexing, and partitions secondary indexes using the partitioning scheme that we refer to as partitioning by term (§3.2.4.2). A difference between SLIK and our design is that in SLIK an index entry stores only data items' primary keys, while in Proteus, index entries also store the data items' attributes. Therefore, in SLIK, an index lookup needs to retrieve each data item in the lookup results from the storage engine.

To achieve consistent index lookups SLIK uses two techniques. First, index entries are created before updates are applied to the corresponding data items, and old index entries are removed asynchronously, after updates have been applied. This ensures that the lifespan of an index entry spans that of the

corresponding data item, and thus guarantees that if a data item contains a secondary attribute value, then an index lookup for that value will return the data item. Second, the query engine removes false positives by verifying index lookup results with the corresponding corpus data items. In other words, corpus data is treated as the ground truth and index entries as hints.

Moreover, SLIK performs long-running bulk operations such as index creation, deletion and migration in the background, in order to avoid blocking blocking normal operations.

While the authors discuss alternative approaches for several aspects of secondary indexing, SLIK's design specific certain points in the design space. For instance, it ensures strong consistency for index lookups at the expense of increased query response time (due to the mechanism that removes false positives). In contrast, the query processing architecture proposed in this thesis aims at occupying different points in the secondary indexing design space.

D'Silva et al. [54] study two approaches for partitioning a secondary index over a distributed data store. The first approach implements indexes as tables in the underlying data store. As a result, index entries are partitioned using the partitioning scheme implemented by the database. In the second approach, each node is responsible for maintaining its portion of the secondary index. In this approach, index entries are stored on the same node as the corresponding base table records. Both approaches have been implemented in HBase [70].

Experimental comparison of the two approaches shows that there is no single right solution in this setting. Rather, each approach is better suited for different needs, depending on factors including the distribution of values of the indexed attribute, the size of the system (number of partitions), the amount of concurrency in index lookups, and the selectivity of queries.

While other works [76, 135] analyze the design alternatives and trade-offs related to secondary indexing in distributed data stores, the work of D'Silva et al. is, to our knowledge, the first to experimentally demonstrate that no single approach to secondary indexing can be best for all cases.

The work of D'Silva et al. considers a system model of a cluster of servers, and employs a synchronous maintenance scheme. Our work expands on that work in two directions. First, we consider geo-distribution. Second, in addition to analyzing secondary indexing design alternatives, we propose an architecture for enabling configurable secondary indexing techniques.

9.1.1 Consistency between corpus and index

In traditional database management systems, search is a primary mechanism for data retrieval. In addition, indexes are used internally for other operations such as view maintenance. Due to these reasons, these systems keep indexes always up to date with corpus data by performing index maintenance synchronously, in the critical path of write operations. In contrast, in non-relational distributed databases there exists a spectrum of design alternatives for index maintenance. In this section, we present an outline of index maintenance approaches proposed in the literature.

Azure DocumentDB is Microsoft's multi-tenant distributed database service for managing JSON documents at large scale. Shukla et al. [130] describe DocumentDB's indexing subsystem. DocumentDB supports two index update policies. In "consistent" mode, the index is updated synchronously as part of the document update, and therefore queries have the same consistency level as the level specified for the point-reads (consistency levels include strong, bounded-staleness, session and eventual). In "lazy" mode, the index is updated asynchronously, when enough resources are available, so that indexing can proceed without affecting the performance guarantees offered for user requests. As a result, queries are eventually consistent (they can observe a stale) This mode is better suited for "ingest now, query later" workloads requiring efficient document ingestion. The indexing policy is set at the level of a document collection.

Diff-Index [135] and Hindex [136] study secondary indexing in data stores based on the log-structured merge tree data structure, which are characterized by the lack in-place update functionality and asymmetric read/write latency. This makes maintaining a consistent index with reasonable update performance challenging.

Diff-Index [135] proposes algorithms for alternative levels of consistency between corpus and indexes: causal, eventual and session consistency. The system enables users to choose the consistency level in a per-index basis. To implement eventual consistency, Diff-index logs writes that require index processing in a queue. Writes are immediately acknowledged to the client, and a background process ingests the queue and updates the index. Session consistency is implemented by tracking additional state in the client library. We note that the authors define causal consistency as the guarantee that once a write operation is acknowledged to the client, both corpus data and associated index entries are persisted in the data store. In this thesis, we use the term strong consistency for this guarantee.

Hindex [136] decomposes the task of index maintenance into two sub-tasks: inserting new index entries and removing old index entries (called index-repair). It executes the fast insert task synchronously while deferring the expensive repair task. Hindex makes use of a compaction mechanism used in log-structured

data stores, which cleans up obsolete data and reorganizes the on-disk data layout. The authors propose two scheduling strategies for the repair operations: offline, coupled with the compaction mechanism, and online, where index-repair operations are performed in the execution path of query operations.

Enabling configurable index (and, more generally, state) maintenance schemes is one of the design goals of the work presented in this thesis. This is achieved by: 1) supporting both synchronous and asynchronous streams between QPUs, and 2) allowing different QPU types to treat updates using different schemes.

9.2 Materialized views

Materialized views were devised for pre-computing the results of expensive query computations in relational databases. Support for materialized views is limited [103] and views must usually be rebuilt when there are changes to the corpus [108].

The concept of materialized views has been employed by a number of works [19, 62, 74] aiming at improving ad-hoc application caching approaches.

Noria [62] is database system that lowers latency and increases supported load for read-heavy applications by using materialized views. It is designed with the aim of delivering the same fast reads as an in-memory cache in front of a database, without the application having to manage the cache. Noria's shares several similarities with our design. Applications register long-term queries with Noria for repeated use. Noria constructs a data-flow graph that connects database tables at its inputs to materialized views at its leaves, and incrementally evaluates these queries when the underlying data changes. Similarly to a QPU-based architecture, Noria's data-flow is a directed acyclic graph of relational operators. This is because both systems' designs are based on database query planning.

Pequod [74] and DBProxy [19] are application-level caches that support incrementally maintained, partially-materialized views [149]. Both systems maintain materialized views by subscribing to a stream of updates propagated by the underlying database. Our design uses the same technique in order to propagate update information from the data store to the QPU graph.

Pequod, similarly to our design, acts as a write-around cache; Application writes go directly to the database, and applications access Pequod only for reads. In contrast, in Noria and DBProxy, applications issue writes to the cache, and the system transparently forwards them to the database.

In Noria and Pequod, view materialization is controlled by users. Noria receives a program specifying a relational schema and a set of parametrized queries, written in SQL, and installs a corresponding data-flow graph. In addition, Noria is able to adapt its data-flow graph to new queries without downtime. Similarly, in Pequod users can textually define views using a simple grammar. In contrast, DBProxy decides dynamically which views to maintain and which can be evicted to save space. It achieves that by transparently intercepting application SQL calls.

There are two main differences between Proteus and these works. First, these works consider the problems of view selection and view definition, and address aspects such as materializing views for additional queries at runtime. In contrast, in this thesis, we focus on the challenges of maintaining materialized views over geo-distributed data. We consider the problems of view selection and view definition to be out of the scope of this work. In our current design, a QPU-graph (akin to Noria's data-flow graph) is defined by the database administrator by passing the corresponding configuration to the graph's query processing units. However, the task of configuration and deployment of a QPU graph based on a given query could be automatically performed by an external component, analogous to Noria's or Pequod's dataflow generation components. The second difference is that Proteus is designed as a more general-purpose query processing framework: in addition to materialized views, Proteus supports caching and secondary indexing.

9.3 Result caching

An important aspect of query result caching is the problem of keeping cached results consistent with the corpus as changes to the corpus occur. Caching systems employ invalidation and refreshing techniques for achieving that. We can categorize cache invalidation and refreshing approaches in two types. *Coupled* or "push-based" approaches provide the cache with information about changes to the corpus. *Decoupled* or "pull-based" approaches invalidate cached entries without any concrete knowledge of changes to the corpus.

Cambazoglu et al. [37] present the design of the result cache used in the Yahoo! search engine. This caching system uses the decoupled approach: Cache entries are invalidated using a time-to-live based strategy, and are refreshed by issuing refresh queries to the web search index. The authors present an algorithm for prioritizing cache entries to be refreshed based on the access frequency of entries and the age of the cached entry.

Proteus supports both push and pull: In pull-based refreshing, the cache QPU refreshes cache entries by sending query requests to the underlying query engine. In push-based refreshing, the cache QPU, upon

a cache miss issues a query request and additionally subscribes to notifications for the corresponding query results. In Proteus, the invalidation and refresh policy of a cache QPU is controlled by a configuration parameter. In addition, the cache QPU supports a hybrid approach in which it receives update notifications, and refreshes the cache entry when a threshold number of updates is crossed, by performing a query request.

9.4 Geo-distributed query processing

A series of papers [23, 35, 36, 59, 75] have studied the problem of designing a large-scale web search engine over multiple geographically distributed data centers.

These works consider the following system model: A search engine architecture composed of multiple data centers, each associated with a geographical region. Each data center stores and crawls documents that are served by web sites in its geographical region. As a result, each data center is responsible for a disjoint subset of the Web. The search engine builds an inverted index over the crawled documents, which is used to serve queries. Documents are ranked based on a ranking function; the result of a query consists of the k most highly ranked documents for that query.

Cambazoglu et al. [35] present a taxonomy of search engine architectures for this problem. In a *centralized* architecture, the entire index is stored in one, central site, and all user queries are submitted to this site. The index might be replicated or partitioned within the site. This architecture was used by early web search engines as well as small-scale engines.

In a *replicated* architecture, the entire index is replicated over multiple data centers. Queries are routed to data centers based on geographical proximity of users to data centers.

Finally, in a *partitioned* architecture, the document collection is partitioned into smaller, non-overlapping sub-collections such that each data center is responsible for a different sub-collection. Queries originating in a particular region are evaluated over the partial index in the corresponding search site. The underlying assumption in this approach is that users are interested more in documents located in their own region, and local documents are more relevant for queries originating from the same region.

This problem is similar to the problem of multi-cloud federated metadata search discussed in Section 6.2. Similarly to the multi-site web search problem, in multi-cloud metadata search a corpus is partitioned into disjoint parts, each placed on a geographically distant data center. Thanks to its flexibility, our architecture design supports all three types of search engine architectures discussed in [35]. This use case is amongst the main motivations for this work.

There are two main differences between the problem examined in this thesis and multi-site web search. First, our query model does not involve the notion of a ranking function. In both problems, the challenge with the partitioned architecture is to retrieve relevant query results that are not local to the site serving a given query. Taking advantage of the ranked query results, approaches to multi-site web search have proposed two solutions: partial replication of the popular documents across search sites [59], and selectively forwarding queries to remote sites that are expected to contribute relevant query results [23, 36]. In this work, we rely on techniques such as caching and replication of popular index and materialized view entries in order to reduce the need to forward queries to remote sites. Cambazoglu et al. [36] have evaluated the impact of result caching on the rate of locally processed queries. Results show that with result caching, the fraction of queries that can be locally processed inc increases by 35% to 45%.

Second, the problem of web search has a different index update model than the problem of attribute-based querying. In web search, indexes are updated periodically based on information from crawlers, while attribute-based querying involves incremental index updates, and potentially high rates of updates.

9.5 Modular & Composable architectures

Click [79] proposes a software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules, called elements. A Click element represents a unit of router processing. Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at its vertices. An edge between two elements represents a possible path for packet; packets flow along the edges of the graph.

While aimed at a different field, Click's core philosophy is close the QPU model. A Click element, akin to a query processing unit, is responsible for a simple packet processing function, and different types of elements are responsible for different packet processing tasks.

Pleiades [29] is a framework for constructing and enforcing large-scale distributed structural invariants. It addressed the need of system architectures to organize nodes in complex topologies. Pleiades is based on assembly-based modularity for enabling configurations to express complex topologies: A structural invariant is expressed as a combination of basic shapes, such as rings, grids, and stars. A configurations specifies a set of basic shapes, and how these shapes should be connected.

9.6 State and computation placement

9.6.1 Computation placement

The problem of the placement of computations on processing nodes has been studied primarily in the context of distributed stream processing systems [86].

A stream processing system can be viewed as a flow graph composed of a set of message sources (producers), and a collection of stream processing operators that periodically consume messages, perform processing tasks, and deliver results either to message sinks (consumers) or to other operators. In addition, there exists a collection of processing nodes available for deploying operators. The operator placement problem consists of computing an assignment of operators to nodes that optimally satisfies a certain metric, such as load, latency, or network resource usage.

From an architecture point of view, placement decisions are performed either by a central placement module, independent from the stream processing engine, or are decentralized, each operator taking local placement decisions. Our work currently consider the placement decisions as independent to the query processing tasks; the placement of the QPU graph on system nodes is decided on by an central entity (a system operator in our current design) before the system is deployed. A direction for future work could be a hybrid placement logic, in which after an initial global placement decision, individual QPUs make local decisions at runtime in response to changing workload, network and resource conditions.

Placement algorithms can be categorized to centralized and decentralized. Centralized placement algorithms require global view of the system, including workload information and resource availability. These algorithms can compute globally optimal placement assignments, but often have limited scalability. On the other had, decentralized algorithms achieve improved scalability by making placement decisions based on local workload and resource information. Our design implicitly assumes that an system operator deciding on the placement of a QPU graph has a global view of the system.

An important difference with these works is that in our current design placement decisions are performed offline, before a QPU graph is deployed, and placement is static throughput the system's operation.

Another distinction is that, most often, this approaches consider stateless operators. In this work we propose an approach for enabling flexible placement of not only stateless operators, such as filters, but stateful components such and indexes and materialized views. In addition, we take into account techniques such as partitioning and replication of query processing state. We consider this one of our mains contributions.

More generally, in this work, we focus on the *mechanisms* (as opposed to placement algorithms) required for enabling flexible placement. While we evaluate different placement configurations in order to demonstrate the benefits of flexile placement, we consider placement algorithms an orthogonal problem that is out of the scope of this work.

Two optimization techniques often employed by operator placement algorithms are operator reuse and replication. Operator reuse is based on the observation that consumers commonly execute identical of partially identical queries. Instead of instantiating new operators for each query, some placement algorithms [106] try to detect opportunities for sharding intermediate results across queries. This avoids transmitting multiple redundant copies of the same data. The design and implementation of query processing units includes two mechanisms that enable operator reuse. First, each QPU can perform multiple independent processing tasks in parallel. As a result, queries that require the functionality of a certain QPU can share the same QPU instance, provided there are enough resources. Second, when a QPU receives a query request, it first examines if that particular query is already being executed, and if so it appends the component that submitted the query to the receivers of the query's output stream.

Operator replication consists of deploying multiple instances of an operator, and partitioning the input stream among operator replicas. This enhances the scalability of stream processing as it parallelizes the processing task among operator replicas [134]. This technique has been also applied to distributed query processing. Query processing in CockroachDB [107] uses the concept of grouping to partition a logical stream (that is part of a query processing task) into multiple physical streams which can be processed in parallel. A stream is divided into groups so that computation within a group is independent from other groups. Parallelizing query processing across operator replicas is also possible in QPU-based architectures. The QPU graph and configuration for achieving that is akin to the index partitioning scheme presented in Section 6.1.

We note that in the the scope of this work neither operator reuse nor replication are automatically handled by the system, as in some stream processing systems. The system however provides the functionalities required for putting in place these techniques. Employing operator replication and reuse in a QPU graph can be performed by configuring the graph accordingly.

Helios [109] is a system used at Microsoft for ingestion, indexing, and aggregation of large streams of real-time data. Helios is used to collect very large amounts (in the order of several PB per day) across 15 data centers. It was developed in response to limitation of existing systems ("None of our existing systems could

handle these requirements adequately; either the solution did not scale or it was too expensive to capture all the desired telemetry.") Amongst Helios' principal design decisions is to support computation offloading at the edge. In Helios' architecture, data collection is performed by an independently-running executable that can run in any pod/rack/data center. Using this components the tasks of data summarization, index generation, and aggregation can be distributed to edge devices.

Helios' support for computation offloading at the edge is similar to the core design goal of the architecture proposed in the work: enabling flexible placement of both (query processing) state and computation across the system. While we have focused on scenarios in which data sources reside in the data center, QPU-based architectures are not restricted to such configurations and can be also used to ingest and index data from geo-distributed sources. Helios is, of course, a system deployed in production in Microsoft and provides support for aspects that are out of the scope of this work, such as the ability to impose restrictions on the resource consumption of the data collection component so that in can run on resource constrained devices.

9.6.2 State placement

Large scale web services are concurrently used by a large number of geographically distributed clients. A significant challenge that these services face is the large request latencies resulting from the geographical distance between clients and application state. A common solution to this latency problem is to place data closer to clients in order to avoid costly round-trips to the data center. For example, Google has comparatively few data center locations relative to edge nodes [6].

Content delivery networks (CDNs), such as Akamai [100] aim at deploying data on edge nodes, close to clients. However, CDNs focus on static data as such images and video content and are not well suited for mutable state.

Other works have studied the problem placing application state across data centers. Volley [14] is a system for automatic data placement across geo-distributed data centers. Volley makes placement decisions based on request logs submitted by applications. The difference between the placement of application state and derived state used for query processing (indexes, materialized) views, is that efficient placement decisions about application state result in reduced latency for both reads and writes to the state. However, in the system model considered in this work, corpus data has a static placement, while derive state can be flexibly placed across the system. This often results to trade-offs between read and write operations, as derived state receives read requests from the clients, and update notifications from the corpus.

9.7 Distributed computation models

9.7.1 MapReduce

MapReduce [47] is a programming paradigm for distributing computational tasks over multiple processing nodes. The core idea is that many types of tasks can be expressed as a map operation that operates on the dataset which is organized as a collection of records (key-value pairs); The map operation can be distributed, each individual operation processing a different subset of the input dataset. The output of these map operations can be then collected and merged into the desired result by reduce operations.

McCreadie et al. [93] have examined the benefits that can be obtained by performing document indexing using MapReduce. In particular, this work presents and evaluates four strategies for applying the MapReduce paradigm to the task of indexing large document collections.

The query processing unit computation model could be compared with the MapReduce paradigm. Two central characteristics of MapReduce are that: 1) each map task is independent and not aware of its context in the overall job (because of this that map tasks can be performed by a large number of workers in parallel, each operating on a different subset of the input dataset) and, 2) map and reduce tasks form a two-tiered hierarchical structure, with the output of the map tasks being used as input to the reduce tasks. The QPU model has similar characteristics: Each individual QPU is independent, and communicates with its connection in the graph graph through well-defined interfaces; Connections between QPUs from from tiered architectures in which the output of one tier is used as input to the next. Because of these similarities, a QPU graph can be configured to perform MapReduce-style distributed computations. For example, the task of indexing a large document collection, can be implemented with a QPU-based architecture as follows: "Indexing" QPUs, responsible for receiving documents as input, can be used as map tasks. For each document, an indexing QPU computed and emits a (term, docID) records. "Reduce" QPUs be connected a parents of multiple indexing QPUs, so that each reduce QPU receives the output of multiple indexing QPUs. Reduce QPU then merge (term, docID) records into posting lists for each docID.

On the other hand, the MapReduce paradigm and the QPU model have important differences in their design goals. MapReduced has been developed with the aim of parallelizing and distributing one-time computational tasks, while the QPU model is mainly intended for building and incrementally maintaining derived state.

9.7.2 Dataflow engines

An evolution of the MapReduce model is the model used in dataflow execution engines. Dataflow engines explicitly model the flow of data through processing stages (henge the term dataflow). A dataflow system represents computations as a graph, in which vertices are user-defined operators. An operator receives input records, process each record and emits zero or more output records. Edge carry records between operators; the output of one operator becomes the input of another. Some of the most well known systems that perform dataflow computations at their core is Apache Spark [148] and Flink [38].

These systems are based on research systems such as Dryad. Dryad [73] is a general-purpose distributed execution engine that employs the dataflow model. A Dryad job is a directed acyclic graph; each vertex is a program and edges represent data channels. Dryad's runtime is responsible for mapping a logical computation graph to physical resources, and abstracts several distributed computing problems from the developer, such as resource allocation, scheduling, and the transient or permanent failures.

Noria [62] proposes partially-stateful dataflow, a dataflow model that supports eviction and reconstruction of data-flow state on demand. Noria employs partially-stateful dataflow to support partially materialized view maintained using updates from a database system.

The QPU computation model also makes use of dataflow computations for maintaining derived state over corpus data. An important difference from general dataflow engines is that the QPU model is designed for executing query processing computations rather than arbitrary distributed computations.

Chapter 10

Future Work and Discussion

10.1 Future Work

In this chapter, we outline directions for future work and discuss additional topics related to the work presented in this thesis.
Directions for future work include:

- Studying algorithms for computing efficient placement of the QPU graph across the system based on application-specified optimization goals and constraints (§10.1.1).
- Extending the current design for supporting online changes to the QPU graph topology and placement, with the aim of dynamically adjusting to workload changes. (§10.1.2).
- Developing protocols for transactionally consistent query processing over eventually consistent derived state (§10.1.3).

The discussion includes the scope and limitations of the work (§10.2.1), fault-tolerance (§10.2.2), the dependence of our design on APIs provided by the storage tier (§10.2.3), and the implications of geo-replication under weak consistency in the maintenance of derived state (§10.2.4).

10.1.1 Placement policies

In this work, we have focused on the mechanisms required for supporting flexible placement of state and computations in geo-distributed query processing. Chapter 4 presents an analysis of the trade-offs involved in placement decisions, and Chapter 8 presents an evaluation of the effects of placement decisions on query processing performance and query result freshness. However, our design relies on developers or system administrators for making placement decisions. An important direction for future work is to study query processing state placement algorithms. This can be expressed by the following problem statement: Given a QPU graph (or more generally, a directed acyclic graph of stateful and stateless query processing operators), a collection of processing nodes available for placing operators organized in a hierarchy of clusters and data centers, and a description of the placement of corpus data on processing nodes, how can we compute an assignment of operators to processing nodes that optimally satisfies an optimization criterion. This is a challenging problem because the inherent tensions between different optimization criteria mean that optimizing for a given metrics incurs penalties to other metrics. However, applications are rarely interested in a single metric. Furthermore, placement decisions need to take into account the applications' workload characteristics.

10.1.2 Dynamic query engine architectures

Throughout this thesis, we make the assumption that QPU graphs are static. Extending the current design with mechanisms and policies for online reconfiguration of QPU graphs, with the aim of adapting to changes in the workload, is an interesting area for future investigation. There are several directions to examine:

- When a query processing unit that maintains an index or materialized view becomes too loaded, or the size of its state grows beyond a given threshold, the unit could decide to split its state into shards by spawning child QPUs, and offloading parts of its state to them. Conversely, when shards are underutilized, they could decide to be merged into a larger shard in order to reduce resource consumption. This requires extending query processing units with support for deploying other units, modifying their configuration during runtime, and modifying the QPU graph topology during runtime.

- Static placement strategies cannot respond to changing workloads. Adjusting the QPU graph's placement to workload changes requires the ability to migrate units. Employing existing state migration techniques in this work should not pose significant challenges.
- This thesis takes a simple approach in the area of the QPU architecture's query processing capabilities. A QPU graph is defined manually, in order to provide support for a pre-determined set of queries. Adding new queries, or materializing additional views or indexes requires manual reconfiguration of the graph, or deployment of a new graph. Interesting future work directions include techniques for automatically generating QPU graphs based on a given set of queries, adding new queries and materializing views and indexes on the fly. Existing works have proposed approaches for generating query processing operator graphs based on user-specified queries [62, 74]. Extending this work with such techniques would require addressing the problems described above, such as supporting online reconfiguration of the QPU graph.

10.1.3 Consistent distributed queries

In Proteus' implementation of the *Query*() interface, a query that requests the latest state snapshot is served from the most recent versions available at each stateful QPU in the query's execution path, in a best-effort fashion. When combined with asynchronous propagation of updates through the QPU graph, this can result in cases in which a query observes different snapshots at different QPUs. For example, consider that case of a QPU graph that maintains two materialized views. An application performs a write operation that will eventually modify both views, and subsequently issues a query that requires reading from both views. The write operation is propagated to one of the views, but the message to the other view is delayed, and, as a result, the query observes a snapshot that contains the write operation in one view, and a snapshot that does not contain it in the other. This can potentially lead to inconsistencies in query results, cause by computing queries using materialized views with different views of the corpus.

This problem is similar to that of distributed transactions: A query needs to be executed over a consistent snapshot across multiple indexes and materialized views (or index/view shards) that are updated asynchronously.

Our design already provides a mechanism that could be built upon to provide consistent snapshots across QPUs: stateful QPUs maintain versioned state, and queries can specify a snapshot on which to be executed by providing a timestamp. A possible solution could use an additional mechanism for determining the most recent snapshot that has been observed by all QPUs that take part in the execution of a given query (stable snapshot). This snapshot would be identified at the start of the execution of a query, and then sub-queries in the query's execution would use the corresponding timestamp to request that snapshot.

An approach for identifying the most recent stable snapshot could be to periodically exchange metadata between query processing units in the background; Each unit would maintain a view of which updates other units have received and applied, based on these metadata. This mechanism can benefit from the hierarchical nature of the QPU graph: each QPU can summarize the view of its outgoing connections, and only propagate that summary to its parents.

10.2 Discussion

10.2.1 Scope

The query engine architecture presented in this thesis is not designed as a general-purpose query engine. Instead, our design assumption is that each QPU architecture is designed for a specific application, with a specific set of queries in mind.

Proteus, the framework that implements the query processing unit architecture pattern is well-suited for applications that:

- Are query-heavy. Our design centers around the use of derived state (indexes, materialized views) for speeding up query processing.
- Have geo-distributed state, or geo-distributed query sources. This work focuses on the area geo-distributed query processing. A large body of research has focused on improving query processing performance in systems that do not involve wide area latencies.
- Use storage systems that lack query processing capabilities, or do not support derived state. Proteus' support for integration with existing storage systems, allows applications to use it as a query processing or derived state middleware.

Furthermore, the query engine architecture presented in this thesis can be applied in the design of database query processing systems that aim to support flexible placement of derived state.

10.2.2 Fault Tolerance

Proteus' approach to fault-tolerance is based on redundancy. When a QPU along the execution path of a query is not available, or fails during the execution of a query, sub-queries to that QPU are retried, and, after a timeout, a failure response is returned to the client. If a QPU can send a sub-query to multiple downstream connections, initially one is chosen, and if that sub-query fails, then the QPU retries by sending the sub-query to a different downstream connection. This mechanism can be used to construct query engines with redundant query execution paths. For example, in the case of a materialized view, a secondary execution path that computed queries without accessing the view can be used as a secondary path in case the materialized view fails.

10.2.3 Data storage tier APIs

The query engine design presented in this thesis is based on the assumption that the data storage tier exposes two interfaces (Section 2.1.1.3): An interface for iterating over the corpus data (list), and one for subscribing to notifications for changes to the corpus data (subscribe). The list interface is used for evaluating queries over the corpus by executing query processing operators over data streams created by iterating over the corpus; the subscribe interface is used for keeping derived state up-to-date with changes to the corpus.

While it is realistic to assume that most storage systems provide a list interface, or interfaces that can be composed to achieve an equivalent functionality, that is not the case for the subscribe interface. Indeed, we have extended AntidoteDB and CloudServer (§7.5) with Proteus-specific notification mechanisms, as such mechanisms were not initially available in these systems.

Proteus could be used with storage systems supporting a subset of the required interfaces. However, this would result in some of the capabilities presented in this thesis not being available. More specifically, if used with a data storage system that supports only the list interface, derived state QPUs would not support incremental updates. Instead, those QPUs would need to periodically rebuild their state by scanning the corpus using the list interface. The query processing unit design already supports periodic rebuilding of derived state.

10.2.4 Derived state in geo-replicated databases

A consideration related to maintaining derived state in geo-replicated databases stems from the CAP impossibility theorem [58]. In the face of network partitions between data centers, geo-replicated databases must choose between strongly consistent and highly-available designs. In a strongly-consistent design, corpus replicas are kept up to date at all times. This provides the abstraction of a single replica system, which simplifies application logic, but exposes users to high write response time due to the need for inter-data center communication between replicas in the critical path of each write operation. In a highly-available design, a corpus replica serves write operations locally and synchronizes with other replicas in the background, out of the critical path of write operations. This ensures low write response time and availability under network partitions, but causes replicas to accept concurrent conflicting updates and to temporarily diverge. Highly-available designs use mechanisms for ensuring that replicas eventually converge to the same state despite conflicting updates, such as conflict-free data types [129]. In some cases, these mechanisms modify the effect of an already processed operation when a concurrent conflicting operation is received later by the replica.

As an example, we can consider the case of an add-wins set [128]. A replica of the set receives an operation o_r that removes an element e from the set, updates its state, and modifies a secondary index accordingly. Later, the replica receives an operation o_a that adds e to the set, and determines that o_r was concurrent with o_r. Based on the conflict resolution rule (add-wins) after both operations have been applied, the element must appear in the set. The same result must be reflected in the index. In contrast, in the case of a remove-wins set, given the same operations, the system needs to update the index with the reverse result. Therefore, in highly-available designs, derived state maintenance algorithms need to take into account concurrent conflicting updates, and conflict resolution policies, in order to avoid divergence between corpus and derived state. Proteus does not currently provide support for this aspect of derived state maintenance.

Chapter 11

Conclusion

Query processing is a crucial component of data serving systems. Nowadays, applications distribute data across multiple geographically distributed data centers in order to efficiently serve users worldwide. Moreover, organizations spread their data and processing between on-premise and cloud environments, as well as between multi-cloud cloud providers, in order to improve fault tolerance and decrease costs. As a result, efficient geo-distributed query processing is essential for addressing the needs of today's internet-scale applications.

In this thesis, we have studied the design decisions and trade-offs involved in the design of geo-distributed query engines that maintain derived state for speeding-up query processing. We have shown that, in the presence of these trade-offs, the placement of query processing state across the system, and the communication patterns involved in query processing and state maintenance are crucial aspects that affect the query engine's performance, effectiveness and operational costs. However, existing systems lack support for configurable placement of query processing state. To address this problem, this thesis has presented a query engine architecture model aimed at enabling flexible and configurable placement of query processing state and computations, and an implementation of that model in Proteus, a framework for constructing application-specific, geo-distributed query engines. The core contribution of this thesis is a query processing component abstraction that combines microservice and stream processing semantics, called Query Processing Unit. This abstraction serves as the building block for composing modular query engine architectures. The characteristics of the query processing unit enable flexibility in the design of the query engine's architecture, and the placement of the query engine's state across the system. This flexibility is essential for navigating the trade-offs of geo-distributed query processing, and adjusting to the requirements and characteristics of different applications.

Bibliography

[1] AWS Pricing Calculator. http://calculator.aws/, .

[2] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/, .

[3] AWS Documentation. Amazon Simple Storage Service (S3): Configuring Amazon S3 event notifications. https://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html, .

[4] AWS Documentation. Amazon Simple Storage Service (S3): Object tagging. https://docs.aws.amazon.com/AmazonS3/latest/dev/object-tagging.html, .

[5] BSON specification. http://bsonspec.org/.

[6] Google Edge Network. https://peering.google.com/#/infrastructure. (visited on 26/12/2020).

[7] Memcached Wiki. https://github.com/memcached/memcached/wiki.

[8] RethinkDB Documentation: Table joins in RethinkDB. https://rethinkdb.com/docs/table-joins/.

[9] Akamai: Consumer Response to Travel Site Performance. https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp, 2010. (visited on 26/12/2020).

[10] InfoWorld. Calculating the true cost of cloud outages. https://www.infoworld.com/article/2613537/calculating-the-true-cost-of-cloud-outages.html, 2013. (visited on 26/12/2020).

[11] Docker Compose file format. https://docs.docker.com/compose/compose-file/, 2020.

[12] Docker Swarm. https://docs.docker.com/engine/swarm/, 2020.

[13] Yaml Ain't Markup Language. https://yaml.org/, 2020.

[14] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2010.

[15] Divy Agrawal, Amr El Abbadi, and Kenneth Salem. A taxonomy of partitioned replicated cloud-based database systems. *IEEE Data Eng. Bull.*, 2015.

[16] Amazon. Amazon Athena. https://aws.amazon.com/athena/, .

[17] Amazon. Amazon Aurora. https://aws.amazon.com/rds/aurora/, .

[18] Amazon. Riak KV 2.2.3 Release Notes: Secondary Indexes Reference. https://docs.riak.com/riak/kv/latest/using/reference/secondary-indexes/index.html, 2017. (visited on 25/12/2020).

[19] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. DBProxy: a dynamic data cache for web applications. In *Proceedings 19th International Conference on Data Engineering*, pages 821–831, 2003.

[20] Apache HBase Team. Apache HBase Reference Guide: Secondary Indexes and Alternate Query Paths. http://hbase.apache.org/book.html#secondary.indexes, 2020.

[21] Apache Software Foundation. Apache Phoenix: Secondary Indexing. https://phoenix.apache.org/secondary_indexing.html, 2020.

[22] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 331–343, 2017.

[23] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vassilis Plachouras, and Luca Telloli. On the Feasibility of Multi-Site Web Search Engines. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 425–434, 2009.

[24] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment*, 2013.

[25] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[26] Matteo Bertozzi. Apache HBase I/O HFile. https://blog.cloudera.com/apache-hbase-i-o-hfile/, 2012. (visited on 25/12/2020).

[27] Dhruba Borthakur. The History of RocksDB. http://rocksdb.blogspot.com/2013/11/the-history-of-rocksdb.html, 2013. (visited on 25/12/2020).

[28] Anna Bouch, Allan Kuchinsky, and Nina T. Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the CHI 2000 Conference on Human factors in computing systems*, 2000.

[29] S. Bouget, Y. Bromberg, A. Luxey, and F. Taiani. Pleiades: Distributed structural invariants at scale. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 542–553, 2018.

[30] Mokrane Bouzeghoub and Verónika Peralta. A framework for analysis of data freshness. In *IQIS 2004, International Workshop on Information Quality in Information Systems*, 2004.

[31] Laura Bright and Louiqa Raschid. Using latency-recency profiles for data delivery on the web. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB*, 2002.

[32] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. Earlybird: Real-time search at twitter. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pages 1360–1369. IEEE Computer Society, 2012.

[33] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.

[34] Mark Callaghan. The advantages of an LSM vs a B-Tree. http://smalldatum.blogspot.com/2016/01/summary-of-advantages-of-lsm-vs-b-tree.html, 2016. (visited on 25/12/2020).

[35] B Barla Cambazoglu, Vassilis Plachouras, and Ricardo Baeza-Yates. Quantifying Performance and Quality Gains in Distributed Web Search Engines. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 411–418, 2009.

[36] B. Barla Cambazoglu, Emre Varol, Enver Kayaaslan, Cevdet Aykanat, and Ricardo Baeza-Yates. Query Forwarding in Geographically Distributed Search Engines. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 90–97, 2010.

[37] Berkant Barla Cambazoglu, Flavio P. Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A Refreshing Perspective of Search Engine Caching. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 181–190, 2010.

[38] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36, 2015.

[39] Irina Ceaparu, Jonathan Lazar, Katie Bessière, John P. Robinson, and Ben Shneiderman. Determining Causes and Severity of End-User Frustration. *International Journal of Human-Computer Interaction*, 2004.

[40] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[41] Surajit Chaudhuri and Vivek R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria*, 2007.

[42] Junghoo Cho and Hector Garcia-Molina. Synchronizing a Database to Improve Freshness. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[43] Google Cloud. Overview of Cloud Bigtable. https://cloud.google.com/bigtable/docs/overview, 2020.

[44] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[45] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[46] Brian F. Cooper, P. P. S. Narayan, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts to Sherpa: Lessons from Yahoo!'s Cloud Database. *Proc. VLDB Endow.*, 12(12):2300–2307, August 2019.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, September 2004.

[48] Jeffrey Dean and Sanjay Ghemawat. LevelDB Implementation Notes. https://github.com/google/leveldb/blob/a0191e5563b7a6c24b39edcbdbff29e602e0acfc/doc/impl.md, 2016.

[49] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[50] AntidoteDB Developers. AntidoteDB Documentation. https://antidotedb.gitbook.io/documentation/, .

[51] Lobsters Developers. Lobsters News Aggregator. https://lobste.rs/, .

[52] Lobsters Developers. Lobsters Rails Project. https://github.com/lobsters/lobsters, .

[53] Lobsters Developers. Lobsters Database Schema (schema.rb). https://github.com/lobsters/lobsters/blob/df5b1e0487094405af196b6322dc09192f27ee5b/db/schema.rb#L195, August 2020.

[54] Joseph Vinish D'silva, Roger Ruiz-Carrillo, Cong Yu, Muhammad Yousuf Ahmad, and Bettina Kemme. Secondary Indexing Techniques for Key-Value Stores: Two Rings To Rule Them All. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference*, 2017.

[55] Elastic. Elasticsearch Reference [7.9]: Scalability and resilience: clusters, nodes, and shards. https://www.elastic.co/guide/en/elasticsearch/reference/7.9/scalability.html, 2020.

[56] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, 1978.

[57] Apache Software Foundation. Apache CouchDB 3.1.0 Documentation: Joins With Views. http://docs.couchdb.com/en/latest/ddocs/views/joins.html, 2020.

[58] Armando Fox and Eric A. Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of The Seventh Workshop on Hot Topics in Operating Systems, HotOS-VII*, 1999.

[59] Guillem Francès, Xiao Bai, B. Barla Cambazoglu, and Ricardo Baeza-Yates. Improving the Efficiency of Multi-Site Web Search Engines. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 3–12, 2014.

[60] Avigdor Gal. Obsolescent Materialized Views in Query Processing of Enterprise Information Systems. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management*, 1999.

[61] Lars George. HBase vs. BigTable Comparison. http://www.larsgeorge.com/2009/11/hbase-vs-bigtable-comparison.html, 2009. (visited on 25/12/2020).

[62] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[63] Google. Protocol Buffers: Language Guide. https://developers.google.com/protocol-buffers/docs/overview.

[64] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[65] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2): 73–169, June 1993.

[66] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. volume 25, pages 173–182, June 1996.

[67] gRPC Authors. gRPC Documentation. https://grpc.io/docs/.

[68] gRPC Developers. gRPC-Go: Histogram. https://github.com/grpc/grpc-go/blob/master/benchmark/stats/histogram.go#L33.

[69] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.*, 2005.

[70] Apache HBase Team. Apache HBase Reference Guide. https://hbase.apache.org/book.html, 2020. (visited on 5/1/2021).

[71] Apache HBase Team. Apache HBase Reference Guide: Regions. https://hbase.apache.org/2.2/book.html#regions.arch, 2020. (visited on 25/12/2020).

[72] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[73] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys)*, pages 59–72, 2007.

[74] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. Easy Freshness with Pequod Cache Joins. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 415–428, April 2014.

[75] Enver Kayaaslan, B. Barla Cambazoglu, and Cevdet Aykanat. Document replication strategies for geographically distributed web search engines. *Information Processing & Management*, 49(1):51–66, 2013.

[76] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kejriwal.

[77] Michael Kerrisk. Linux manual page: tc. https://man7.org/linux/man-pages/man8/tc.8.html.

[78] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[79] Eddie Kohler, Robert Tappan Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 2000.

[80] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[81] Alexandros Labrinidis and Nick Roussopoulos. Balancing Performance and Data Freshness in Web Database Servers. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB*, 2003.

[82] Cockroach Labs. CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/, 2020.

[83] Redis Labs. Redis Documentation: Using Redis as an LRU cache. https://redis.io/topics/lru-cache.

[84] Riak Labs. Riak KV Documentation: Clusters. https://docs.riak.com/riak/kv/2.2.3/learn/concepts/clusters.1.html, 2016.

[85] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[86] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing*, 12(6):50–60, November 2008.

[87] Per-Åke Larson and Jingren Zhou. Efficient Maintenance of Materialized Outer-Join Views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE*, 2007.

[88] Ki Yong Lee and Myoung-Ho Kim. Optimizing the incremental maintenance of multiple join views. In *DOLAP 2005, ACM 8th International Workshop on Data Warehousing and OLAP*, 2005.

[89] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[90] Richard Low. The sweet spot for Cassandra secondary indexing. https://web.archive.org/web/20191228133556/http://www.wentnet.com/blog/?p=77, 2013. (visited on 25/12/2020).

[91] Marko Mäkelä. Deep Dive: Innodb Transactions and Write Paths. https://mariadb.org/wp-content/uploads/2018/02/Deep-Dive_-InnoDB-Transactions-and-Write-Paths.pdf, 2018. (visited on 26/12/2020).

[92] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Record*, 2012.

[93] Richard Mccreadie, Craig Macdonald, and Iadh Ounis. MapReduce Indexing Strategies: Studying Scalability and Efficiency. *Inf. Process. Manage.*, 48(5):873–888, 2012.

[94] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, March 2014.

[95] Inc. MongDB. New Hash-based Sharding Feature in MongoDB 2.4. https://www.mongodb.com/blog/post/new-hash-based-sharding-feature-in-mongodb-24, 2013. (visited on 25/12/2020).

[96] Inc. MongDB. MongoDB Documentation: Shards. https://docs.mongodb.com/manual/core/sharded-cluster-shards/, 2020. (visited on 25/12/2020).

[97] Andrew Morgan. Joins and Other Aggregation Enhancements Coming in MongoDB 3.2. https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction, 2015. (visited on 25/12/2020).

[98] Fiona Fui-Hoon Nah. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? In *9th Americas Conference on Information Systems, AMCIS*, 2003.

[99] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2013.

[100] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, 44(3):219, August 2010.

[101] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). 33(4):351–385, June 1996.

[102] Oracle Corporation. MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/, 2020.

[103] Oracle Corporation. MySQL 8.0 FAQ: Views. https://dev.mysql.com/doc/refman/8.0/en/faqs-views.html#faq-mysql-have-materialized-views, 2020.

[104] Jon Gjengset Peter Bhat Harkins. Lobsters Access Pattern Statistics. https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b. (visited on 26/12/2020).

[105] James Phillips. Surprises in our NoSQL adoption survey. https://blog.couchbase.com/nosql-adoption-survey-surprises/, 2014. (visited on 26/12/2020).

[106] P. Pietzuch, J. Shneidman, M. Roussopoulos, J. Ledlie, M. Seltzer, and M. Welsh. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering*, apr 2006.

[107] Raphael Poss and Alex Robinson. CockroachDB: Distributing sql queries. https://github.com/cockroachdb/cockroach/blob/75f0fa82c1a9e9c0010831e21a884a5746861b66/docs/RFCS/20160421_distributed_sql.md, 2020.

[108] The PostgreSQL Global Development Group. PostgreSQL 9.4.26 Documentation: Materialized Views. https://www.postgresql.org/docs/9.3/rules-materializedviews.html, 2020.

[109] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, and Raghu Ramakrishnan. Helios: Hyperscale Indexing for the Cloud & Edge. *Proc. VLDB Endow.*, 13(12):3231–3244, August 2020.

[110] Tom Preston-Werner. Tom's Obvious, Minimal Language. https://toml.io/en/, 2020.

[111] Houliang Qi, Xu Chang, Xingwu Liu, and Li Zha. The Consistency Analysis of Secondary Index on Distributed Ordered Tables. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops IPDPS*, 2017.

[112] Judith Ramsay, Alessandro Barabesi, and Jennifer Preece. A psychological investigation of long retrieval times on the world wide web. *Interacting with Computers*, 1998.

[113] Kazunori Sato. An Inside Look at Google BigQuery. https://cloud.google.com/files/BigQueryTechnicalWP.pdf. (visited on 26/12/2020).

[114] Scality. Zenko Cloudserver. https://github.com/scality/cloudserver/, .

[115] Scality. Customer Case Studies: Bloomberg Media Group. https://www.scality.com/customers/bloomberg/, .

[116] Scality. CloudServer. https://github.com/scality/cloudserver/, .

[117] Scality. Zenko 1.2.0 Documentation: Architecture. https://zenko.readthedocs.io/en/latest/operation/Architecture/index.html, 2020.

[118] Scality. Zenko 1.2.0 Documentation. https://zenko.readthedocs.io/en/latest/index.html, 2020.

[119] Scality. Cloudserver Documentation. https://github.com/scality/cloudserver/blob/9615d51c9e0500db37905dcfd04b17187132eddf/docs/MD_SEARCH.rst, 2020.

[120] Scality. Zenko 1.2.0 Documentation: MongoDB. https://zenko.readthedocs.io/en/latest/operation/Architecture/MongoDB.html, 2020.

[121] Scality. Zenko 1.2.0 Documentation: Out-of-Band Updates. https://github.com/scality/cloudserver/blob/9615d51c9e0500db37905dcfd04b17187132eddf/docs/MD_SEARCH.rst, 2020.

[122] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI) - Volume 3*, 2006.

[123] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 1979.

[124] MariaDB Server Documentation. MariaDB Foundation. https://mariadb.org/documentation/, 2020.

[125] MariaDB Server Documentation: Triggers. MariaDB Foundation. https://mariadb.com/kb/en/triggers/.

[126] MariaDB Server Documentation: User-Defined Functions. MariaDB Foundation. https://mariadb.com/kb/en/user-defined-functions/.

[127] Shane Johnson. MongoDB fails to perform, runs out of gas. https://blog.couchbase.com/mongodb-fails-to-perform-runs-out-of-gas/, 2016. (visited on 26/12/2020).

[128] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, January 2011.

[129] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400. Springer LNCS volume 6976, October 2011.

[130] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin Levandoski, and David Lomet. Schema-Agnostic Indexing with Azure DocumentDB. In *Proceedings of the VLDB Endowment*, September 2015.

[131] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[132] Apache Software Foundation. Apache Solr Reference Guide: Distributed Search with Index Sharding. https://lucene.apache.org/solr/guide/6_6/distributed-search-with-index-sharding.html, 2017.

[133] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 385–400, 2011.

[134] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[135] Wei Tan, Sandeep Tata, Yuzhe Tang, and Liana L. Fong. Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014*, pages 700–711, 2014.

[136] Yuzhe Tang, Arun Iyengar, Wei Tan, Liana Fong, Ling Liu, and Balaji Palanisamy. Deferred Lightweight Indexing for Log-Structured Key-Value Stores. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15, pages 11–20. IEEE Press, 2015.

[137] The Apache Software Foundation. Apache Cassandra Documentation: Architecture, Dynamo. https://cassandra.apache.org/doc/latest/architecture/dynamo.html, 2016.

[138] Zachary Tong. Customizing your document routing. https://www.elastic.co/blog/customizing-your-document-routing/, 2013. (visited on 25/12/2020).

[139] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[140] Dimitrios Vasilas. Losbsters Benchmark Tool. https://github.com/dvasilas/proteus-lobsters-bench.

[141] Dimitrios Vasilas, Marc Shapiro, and Bradley King. A modular design for geo-distributed querying: Work in progress report. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '18, 2018.

[142] Dimitrios Vasilas, Marc Shapiro, Bradley King, and Sara S. Hamouda. Towards application-specific query processing systems. In *36ème Conférence sur la Gestion de Données : Principes, Technologies et Applications (BDA 2020)*, Online, October 2020.

[143] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 449–462, February 2020.

[144] Amazon Web Services. Amazon DynamoDB Documentation: Data Synchronization Between Tables and Global Secondary Indexes. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html.

[145] Amazon Web Services. Amazon S3 API Reference. https://docs.aws.amazon.com/AmazonS3/latest/API/s3-api.pdf#Type_API_Reference, 2006.

[146] Amazon Web Services. Amazon DynamoDB Documentation: Improving Data Access with Secondary Indexes. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html, 2020.

[147] Andy Woods and Daniel Harrison. CockroachDB Blog: Geo-Partitioning: What Global Data Actually Looks Like. https://www.cockroachlabs.com/blog/geo-partitioning-one/, 2018. (visited on 26/12/2020).

[148] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, April 2012.

[149] J. Zhou, P. Larson, J. Goldstein, and L. Ding. Dynamic Materialized Views. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 526–535, 2007.

[150] Jingren Zhou, Paul Larson, and Jonathan Goldstein. Partially Materialized Views. Technical Report MSR-TR-2005-77, June 2005.

[151] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. Lazy Maintenance of Materialized Views. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.

[152] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. Dynamic Materialized Views. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, pages 526–535, 2007.

[153] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.