Processing continuous join queries in qtructured P2P systems

Wenceslao Palma

UNIVERSITE DE NANTES

ÉCOLE DOCTORALE

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE MATHEMATIQUES »

Année 2010

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE NANTES

Discipline : INFORMATIQUE

Présentée
et soutenue publiquement par

Wenceslao Enrique PALMA MUNOZ

Le 18 Juin 2010
à l'UFR Sciences & Techniques, Université de Nantes

Traitement de requêtes de jointures continues dans les systèmes pair-à-pair (P2P) structurés

---

Jury

Président	:	Pr. Luc Bougé	ENS Cachan
Rapporteurs	:	Luc Bouganim, DR INRIA	INRIA Paris-Rocquencourt
		Pierre Sens, Pr	UPMC Paris 6
Examinateurs	:	Reza Akbarinia, CR INRIA	INRIA Rennes
		Esther Pacitti, Pr	Université de Montpellier 2, LIRMM
		Guillaume Raschia, MC	Polytech'Nantes.
		Patrick Valduriez, DR INRIA	INRIA Sophia Antipolis

Directeur de thèse : Patrick Valduriez
Co-Directrice de thèse : Esther Pacitti

Laboratoire: **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinire, BP 92 208 – 44 322 Nantes, CEDEX 3. N° ED

Traitement de requêtes de jointures continues dans les systèmes pair-à-pair (P2P) structurés

---

Wenceslao Enrique Palma Munoz

$\bowtie$

favet neptunus eunti

Université de Nantes

Wenceslao Enrique Palma Munoz

Traitement de requêtes de jointures continues dans les systèmes pair-à-pair (P2P) structurés

IV+xiv+100 p.

This document was edited with these-LINA LaTeX2e class of the "Association of Young Researchers on Computer Science (IRGIN)" from the University of Nantes (available on : http://login.irin.sciences.univ-nantes.fr/). This LaTeX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies (circulaire n° 05-094 du 29 March 2005) of the University of Nantes and the Doctoral School of « Technologies de l'Information et des Matériaux(ED-STIM) ».

Print : thesis.tex – 11/05/2010 – 5:08.

Abstract

Recent years have witnessed the growth of a new class of data-intensive applications that do not fit the DBMS data model and querying paradigm. Instead, the data arrive at high speeds taking the form of an unbounded sequence of values (data streams) and queries run continuously returning new results as new data arrive. In these applications, data streams from external sources flow into a Data Stream Management System (DSMS) where they are processed by different operators. Many applications share the same need for processing data streams in a continuous fashion. For most distributed streaming applications, the centralized processing of continuous queries over distributed data is simply not viable. This research addresses the problem of computing continuous join queries over distributed data streams. We present a new method, called DHTJoin that exploits the power of a Distributed Hash Table (DHT) combining hash-based placement of tuples and dissemination of queries by exploiting the embedded trees in the underlying DHT, thereby incuring little overhead. Unlike state of the art solutions that index all data, DHTJoin identifies, using query predicates, a subset of tuples in order to index the data required by the user's queries, thus reducing network traffic. DHTJoin tackles the dynamic behavior of DHT networks during query execution and dissemination of queries. We provide a performance evaluation of DHTJoin which shows that it can achieve significant performance gains in terms of network traffic.

Keywords: P2P Systems, Continuous Query Processing

Résumé

De nombreuses applications distribuées partagent la même nécessité de traiter des flux de données de façon continue, par ex. la surveillance de réseau ou la gestion de réseaux de capteurs. Dans ce contexte, un problème important et difficile concerne le traitement de requêtes continues de jointure qui nécessite de maintenir une fenêtre glissante sur les données la plus grande possible, afin de produire le plus possible de résultats probants. Dans cette thèse, nous proposons une nouvelle méthode pair-à-pair, DHTJoin, qui tire parti d'une Table de Hachage Distribuée (DHT) pour augmenter la taille de la fenêtre glissante en partitionnant les flux sur un grand nombre de noeuds. Contrairement aux solutions concurrentes qui indexent tout les tuples des flux, DHTJoin n'indexe que les tuples requis pour les requêtes et exploite, de façon complémentaire, la dissémination de requêtes. DHTJoin traite aussi le problème de la dynamicité des noeuds, qui peuvent quitter le système ou tomber en panne pendant l'exécution. Notre évaluation de performances montre que DHTJoin apporte une réduction importante du trafic réseau, par rapport aux méthodes concurrentes.

Mots-clés : Systèmes pair-à-pair, Traitement de requêtes

acm Classification

Categories and Subject Descriptors : H.2.4 [**Database Management**]: Systems—*Distributed databases, Query processing.*

Acknowledgements

Table of Contents

List of Tables

— Body of the Dissertation —

List of Figures

— Body of the Dissertation —

List of Examples

— Body of the Dissertation —

Résumé Étendu

Chapter 1

Introduction

In recent years, an increasing number of data-intensive applications has emerged. These applications requires on-line continuous processing of unbounded sequences of data that arrive at high speeds (referred to as *data stream*) making space and execution-time performance a critical issue. As has been noted [BBD+02][GÖ03a], traditional database management systems (DBMS) are not adequate for data stream processing. In response to these applications, that do not fit the DBMS data model and querying paradigm, a new class of data management systems, commonly known as data stream management systems (DSMS), has emerged. However, high stream input rates and cost-intensive query operations may cause a continuous query system to run out of resources.

Due to the success of filesharing applications, peer to peer (P2P) networks have received rapid and widespread deployment. Self-organization, symmetric communication and distributed control are attractive properties of P2P networks that motivated researchers to study complex query processing for supporting advanced P2P applications. Range [DHJ+05], aggregate [ZHC08], ranking [APV06] and join [HHL+03] operators have been proposed under different P2P architectures to process queries in diverse domains ranging from P2P reputation to network monitoring.

Because applications that process streams from different sources are inherently distributed [CG07] and because distribution can be used to improve performance and scalability of a DSMS [TcZ07] we are interested in continuous query processing over P2P networks where self-organization, symmetric communication and distributed control can be used to face the challenges motivated by the processing of data streams.

In this chapter, we first introduce stream processing applications and the data management services that a DSMS must provide. We present P2P networks and its potential in the processing of complex queries. We also motivate the processing of continuous queries in P2P networks. Finally, we conclude with the scope, the contributions and outline of this thesis.

1.1 Data Stream Management Systems

Traditional database management systems (DBMS) are powerful for querying stores of persistent data sets. In DBMS the data consist of a set of unordered objects stored as tables with insertions, updating and deletions occurring less frequently than queries. Significant portions of the data are queried again and again by executing *one-time* queries which are run once to completion over the current data sets stored in disk (see Figure 1.1). Recent years have witnessed the growth of a new class of data-intensive applications that do not fit the DBMS data model and querying paradigm. Instead, the data arrive at high speeds taking the form of an unbounded sequence of values and queries run continuously over these *streams* returning new results as new data arrive (see Figure 1.1). In these applications, data streams from external sources flow into

a data stream management system (DSMS) where they are processed by different operators. Examples of these applications include network monitoring, sensor networks, financial analysis applications and analysis of transactional log data :

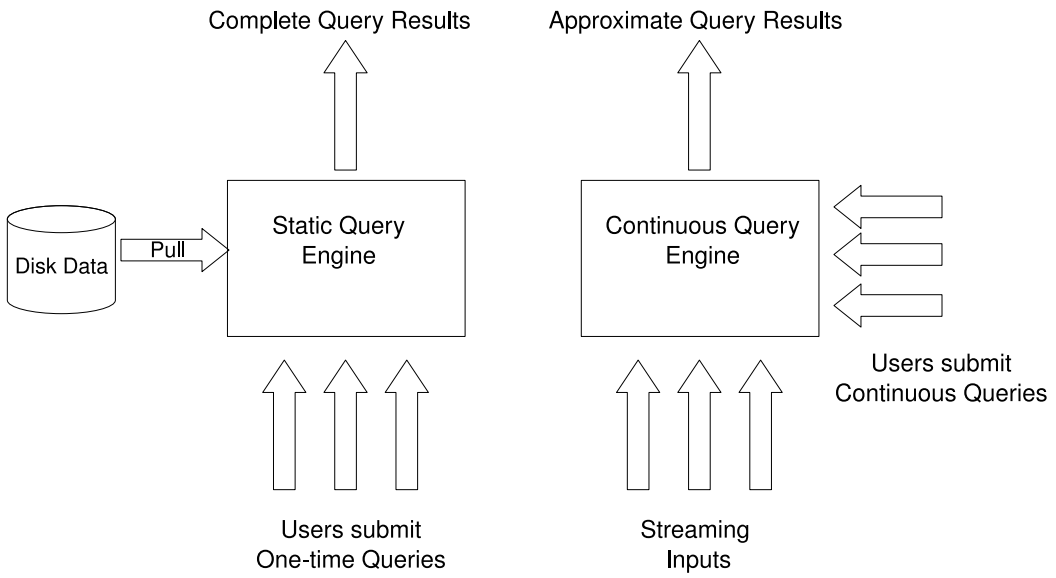


Figure 1.1 – Traditional DBSM vs Continuous DSMS

- In the networking community, network operators and service providers need to monitor network traffic [Sul96][CJSS03] to analyze the provided service level, detect suspicious activity, identify bottlenecks, etc. In this context, the streams arrive at high speeds and are composed of well-structured IP packets that can therefore be queried continuously to compute statistics over these streams.
- Sensors networks with wireless capabilities are being deployed for sensing the physical world. Sensors continuously deliver data in *streams* for measurement [YG03], detection and surveillance applications [ACG+04]. In these applications users issue long-running queries over the streamed data.
- In electronic trading markets [LS03] such as the NASDAQ, large volume of data streams are generated at speeds that can achieve up to tens of thousands of messages. In this context data streams represents momentary opportunities to make profits if they are processed efficiently. A financial analyst can submits continuous queries to detect simple conditions (e.g. when a price is the maximum price of the day) and calculate correlations among price time series.
- On-line analysis of transactions logs, [MAA05][CFPR00] generated by telephone call records and HTTP requests to a web server are very important to discover customer spending patterns. For example, the ATT&T long distance telephone call stream is composed by 300 million of records per day generated from 100 million customers.

In response to the novel requirements imposed by the above applications a DSMS must provide :

- *Persistent queries.* In a stream processing application, queries are persistent in the sense that are issued once but remain active for a long time and must be processed continuously as new input data arrives. In contrast, a traditional DBMS executes *one-time* queries (issued once and then "forgotten") again stored data.
- *Push-based processing.* Contrary to DBMS, the processing of queries is push-based, or data-driven. Data sources continuously produce streams items that are pushed into the system for processing. This contrast with DBMS where the processing is pulled-based or query-driven, i.e., the clients actively pull information about the data when they need it.
- *Approximate answers with emphasis over recent data.* Generally a data stream has a unbounded, or at least unknown, size. This setting imposes high processing and memory requirements. From a system's point of view, it is infeasible to store an entire stream in a DSMS and query plans must avoid blocking operators that must consume the entire input before any results produced [LWZ04]. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurements or utilization patterns. Moreover, in the majority of real world applications emphasizing recent data is more informative and useful than old data.

1.2 P2P networks

P2P networks are distributed system in nature that satisfy roughly the following three criteria [RBR+04] :

- *Self-Organization* : A P2P network automatically adapts to the arrival and departure of nodes. There is no global directory of peers and resources. Nodes organize themselves based on local information and interacting with locally reachable nodes (neighbors) and a global behaviour emerges as a result of all the local interactions that occur.
- *Symmetric communication* : Nodes are considered equals. There is no a notion of clients or servers, peers act as both.
- *Distributed Control* : There is no longer a node that centrally coordinates the behaviour of nodes. Nodes determines their level of participation and their course of action autonomously. This decentralization is in particular interesting in order to avoid single-point-of failures or performance bottlenecks.

In general, there are three P2P architectures : unstructured, structured and super-peer [VP04]. In unstructured P2P networks, each node can directly communicate with its neighbors. The peers use flooding as the mechanism to send queries across the network with limited scope. When a peer receives a query, it processes and redirects the incoming query to its neighbors. The flooding mechanism is effective for locating high replicated data objects and provides resilience to nodes joining and leaving the system. However, it does not scale up to large number of nodes.

Under structured P2P networks, peers are organized according to specific topologies (e.g., ring, hypercubes, tree-like, butterfly and others) based on a certain order [GGG+03]. The network topology is tightly controlled and data are placed not at random nodes but in a specific way to facilitate query processing. Structured P2P networks use the Distribute Hash Table (DHT) as a substrate. Data objects and nodes are assigned unique identifiers called *keys* and data objects are placed at nodes with identifiers corresponding to the data object's unique *key*. DHTs offer a good scalability (typically as $log n$) and has been proposed as a substrate for many large-scale distributed applications.

Super-peer P2P networks employs more powerful nodes as *super-peers*. A *super-peer* is a node that acts as a centralized server to a subset of peers that act as dedicated server and can perform complex functions such as indexing, query processing, access control and meta-data management. Peers submit queries to their *super-peer* and receive results from it. The main advantage of super-peer P2P networks is efficiency and quality of service. However, fault tolerance is typically low since a *super-peer* is a single point of failure.

The success of P2P applications and protocols as DHTs [SMK+01] motivated researchers from the distributed systems, networking and database communities to look more closely into the core mechanisms of P2P networks and investigate how these could be used to support another applications than filesharing. For example, under the different P2P architectures many research works propose query operators to support various functionalities :

- *Range query operator* is important in supporting complex structured database-like queries. However, *keys* which are semantically close at the application level are heavily fragmented in a DHT. Many approaches [ACMD+03] [RRHS04][BAS04][JOV05] has been proposed in order to implements efficiently range queries in structured P2P networks.
- *Aggregate query operator* is used to evaluate global states in P2P networks [ADGK07] [JMB05] [BGMGM03]. For example, calculating the global reputation by aggregating peers feedback [XL04] [ZHC08] is very important in P2P reputation systems where aggregate queries are needed for quantifying and comparing the trustworthiness of participating peers to combat malicious peers behaviours (e.g., dishonest feedback).
- *Ranking query operator* is a crucial requirement in many environments that involves massive amount of data. In many domain applications users are interested in the most important (top-k) query answers in the potentially huge answer space. The main reason is that a ranking query operator avoids overwhelming the user with large numbers of uninteresting answers which are resource consuming. In unstructured P2P networks top-k queries are proposed to rank query results based on user preferences and in this way reduce the heavy network traffic [APV06][ADGK07].
- *Join query operator* is important in P2P networks where data normally originate from multiple sources and the naive solution of collecting all data at a single site is simply not viable [CG07]. In many applications such as network monitoring [CCD+03], sensor networks [BGS01] the join operator is one of the most important operators, which can be used to detect trends between different data sources. In [BT03] a P2P DBMS called AmbientDB is developed to process queries over and ad-hoc P2P of many, possibly small devices. In AmbientDB, DHTs are used to support efficient global indices that can transparently accelerate queries (e.g. join queries). Another efforts of designing complex querying facilities focusing in join queries are [HHL+03] and [ILK08].

1.3 Problem Statement

Processing a query over a data stream involves running the query continuously over the data stream and generating a new answer each time a new data item arrives. However, the unbounded nature of data streams makes it impossible to store the data entirely in bounded memory. This makes difficult the processing of queries that need to compare each new arriving data with past ones. For example, real data traces of IP packets from an AT&T data source [GJK+08] show an average data rate of approximately 400 Mbits/sec, which makes it hard to keep pace for a DSMS.

Moreover, a DSMS may have to process hundreds of user queries over multiple data sources. For most distributed streaming applications, the naive solution of collecting all data at a single site is simply not viable [CG07]. Therefore, we are interested in techniques for processing continuous queries over collections of distributed data streams. This setting imposes high processing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurements or utilization patterns.

One technique for producing an approximate answer to a continuous query is to execute the query over a sliding window [GÖ03b] that maintains a restricted number of recent data items. This allows queries to be executed in finite memory, in an incremental manner by generating new answers each time a new data item arrives. Moreover, in the majority of real world applications emphasizing recent data is more informative and useful than old data.

In continuous query processing the join operator is one of the most important operators, which can be used to detect trends between different data streams. For example, consider a network monitoring application that needs to issue a join query over traffic traces from various links, in order to monitor the total traffic that passes through three routers (R_1, R_2 and R_3) and has the same destination host within the last 10 minutes. Data collected from the routers generate streams S_1,S_2 and S_3. The content of each stream tuple contains a packet destination, the packet size and possibly other information. This query can be posed using a declarative language such as CQL [AW04], a relational query language for data streams, as follows :

q : Select sum (S_1.size)
From S_1[range 10 min], S_2[range 10 min], S_3[range 10 min]
Where S_1.dest=S_2.dest and S_2.dest=S_3.dest

To emphasizes in recent data the window conceptually slides over the input streams giving raise to a type of join called sliding window join. To improve performance and scalability, distributed processing of data streams is a well accepted approach. Self-organization, symmetric communication and distributed control are characteristics present in P2P networks that provide a good substrate for creating distributed applications. However, even in a distributed setting high stream arrival rates and cost-intensive query operations may cause a DSMS to run out of resources. For example, when the memory allocated to maintain the state of a query is not sufficient to keep the window size entirely, the completeness, i.e., the fraction of results produced by an query operator over the total results (which could be produced under perfect conditions) is reduced.

Moreover, in distributed processing of data streams, node failures can occur. The failures can disrupt stream processing affecting the correctness of the results and in some cases it can prevent the system for producing any results.

Considering the problems that are originated during the distributed processing of data streams, the purpose of this research is to take the advantages of P2P networks and to provide efficient processing of continuous join queries.

1.4 Scope, Contributions and Organization

This research addresses continuous join queries over structured P2P networks (see Figure 1.2). This research excludes unstructured P2P networks because they are inefficient in terms of

response time, consume much network traffic and do not provide guarantees of any kind. Join query processing has been well addressed in prior research works [Val87][Has95][LR05] but the unbounded nature of data stream and the P2P environment creates new challenges. We focus in query execution where a DHT is used for routing tuples and as a hash table for storing tuples. However, hash-based redistribution of tuples on their joining attribute is vulnerable to the presence of skew in the underlying data. Moreover, in P2P networks nodes can fail disrupting the execution of queries.

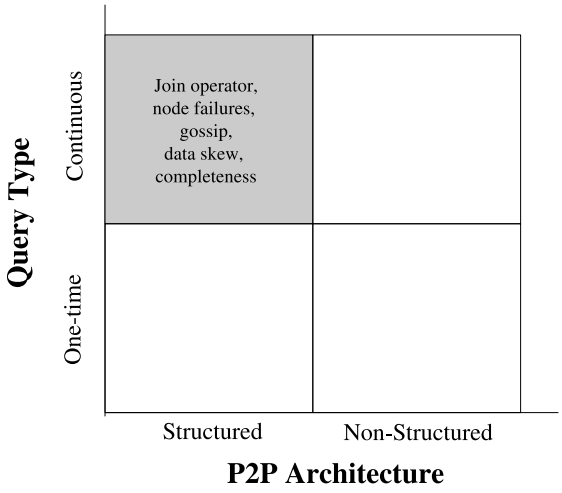


Figure 1.2 – The scope of this research.

We make profit of gossip-based protocols to cope with failures because they are resilient to failures, frequent changes and they cope well with the dynamic behaviour of nodes in P2P networks. Due to the nature of data stream applications, approximate answers are well accepted in continuous query processing. Regarding this issue, we tackle the completeness of results using an approach based on a peer level and on a query level that allows to precise how many resources are needed in order to obtain a certain degree of completeness.

Considering the scope of this research, in the next section we gives a quick overview of the proposed approach to addresses continuous join queries over structured P2P networks and we precise its contributions.

Contributions

In this research we propose an efficient method named DHTJoin to process continuous join queries using structured P2P networks. We provide an architecture for the deployment of DHT-Join in a structured P2P network. This architecture allows the parallelization of continuous join queries using a combination of partitioned and pipelined parallelism. The execution of continuous join queries in DHTJoin is performed by a combination of hash-based placement of tuples and dissemination of queries by exploiting the embedded trees in the underlying DHT. DHTJoin also deals with failures during query execution and query dissemination, and attribute value skew which may hurt load balancing and result completeness. More precisely, the contributions

of this research are following :

- DHTJoin identifies, using query plans, a subset of tuples in order to index the data required by the user's queries, thus reducing network traffic. This is more efficient than the approaches based on structured P2P overlays, e.g. PIER [HHL+03] and RJoin [ILK08], which typically index all tuples in the network. Furthermore, our approach dynamically indexes tuples based on new attributes when new submitted queries contain different predicates.
- We provide an analytical evaluation of the best number of nodes to obtain a certain degree of completeness given a continuous join query.
- DHTJoin tackles the dynamic behavior of DHT networks during query execution and dissemination of queries. When nodes fail during query dissemination, DHTJoin uses a gossip-based protocol that assures 100% of network coverage. The gossip-based protocol is based on the concept of anti-entropy where nodes exchange messages in order to detect and correct inconsistencies in a system. When nodes fail during query execution, DHTJoin propagates messages to prevent nodes of sending intermediate results that do not contribute to join results, thereby reducing network traffic.
- DHTJoin provides an efficient solution to deal with overloaded nodes as a result of data skew. The key idea is to distribute the tuples of an overloaded node to some underloaded nodes, called *partners*. When a node gets overloaded, DHTJoin discovers partners using information in the routing table and determines what tuples to send them using the concept of domain partitioning. We show that, in this case, DHTJoin incurs only one additional message per joined tuple produced, thus keeping response time low.

Organization

This work is organized as follows. In Chapter 2, we survey data stream processing and query processing in P2P networks. First, we give more insight into many aspects of data stream processing such as query languages, query operators and their implementation. Second, we present the main research project in the area of DSMSs and their primary contributions. Then we present P2P systems identifying their fundamental requirements, challenges and its potential in the processing of complex queries. Finally, we discuss the state of the art and introduce our method to process continuous join queries using structured P2P networks.

Chapter 3 presents the architecture of DHTJoin. After a quick overview, we detail the components of the architecture with its different features and we provides a description of the data flow during query processing.

In Chapter 4, we present DHTJoin, our proposed method to meet the challenges of efficiently executing continuous join queries in a structured P2P network. We define formally the type of join queries tackled and their strategies for parallelization. We describe DHTJoin and how it deals with node failures during query dissemination and query execution. Then, we present an analysis of result completeness which relates memory constraints, stream arrival rates and number of nodes required to obtain a certain degree of result completeness. Finally, we show how DHTJoin deals with data skew.

Chapter 5 presents the performance evaluation of DHTJoin through simulation. We show that our method is competitive w.r.t a complete implementation of RJOIN [ILK08] the most relevant state of the art approach dealing with continuous join queries over structured P2P networks. We also show the effectiveness of the approaches proposed to deal with node failures

during query dissemination and query execution.

Finally, we conclude and highlight future directions of research.

List of Publications

International Journals

- Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez. *DHTJoin : processing continuous join queries using DHT networks*. In Journal of Distributed and Parallel Databases, 26(2-3) :291-317,2009.

International Conferences

- Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez.*Efficient Processing of Continuous Join Queries using Distributed Hash Tables*. In Proceedings of the 14th International Conference on Parallel and Distributed Computing (Euro-Par), pages 632-641, 2007.

International Workshops

- Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez. *Distributed Processing of Continuous Join Queries using DHT Networks*. In Proceedings of the 3th ACM International Workshop on Data Management in Peer-to-Peer Systems (DAMAP), 2009.

National Conferences

- Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez. *Traitement des Requêtes Continues de Jointure dans un Environnement P2P*. In Actes des 25èmes Journées Bases de Données Avancées (BDA), 2009.

CHAPTER 2
State of The Art

This chapter reviews the main works on data stream processing and complex query processing in P2P systems. More specifically, in Section 2.1 we survey query languages, in particular, join operators and their implementation. In Section 2.1.3, we present the main research projects in the area of data stream management systems presenting their primary contributions. Finally, in Section 2.2 we present P2P systems, focusing on how complex query facilities are implemented in DHT-based structured P2P systems.

2.1 Data Stream Processing

Query processing over data streams is a challenging task. Traditional DBMSs are not designed to support continuous queries that are typical of data stream applications. In response, a new class of data management systems commonly known as Data Stream Management Systems (DSMS) has emerged. To support stream processing applications, DSMSs introduce new data models, query languages and operators. In this section, we show how users can express their queries and how query operators are implemented to cope with the requirements of data stream applications. Finally, we survey recent work on DSMS.

2.1.1 Data Model and Query Languages

A data stream is an unbounded append-only sequence of timestamped data items. In relation-based stream models (e.g. STREAM [BBD$^+$02], DCAPE[ZR07], Borealis[AAB$^+$05]) data items take the form of tuples such that all the tuples belonging to the same stream have the same schema, i.e. they have the same set of attributes. Tuples may contain timestamps assigned explicitly by its source or assigned by the DSMS upon arrival. The timestamp may or may not to be part of the stream schema. Tuples are produced continuously by data sources and are pushed into the system and generally stored in main memory where they are processed through the queries. In the majority of real-world streaming applications, analysis based on recent data is more informative and useful than analysis based on stale data. This constraints queries to return results inside a window (e.g. the last 30 minutes or the last hundred of packets). For example, in network applications, emphasizing recent data is more informative and useful than old data. This part of the user's requirements is expressed as part of the user query, therefore it must be supported by the query language used to express the queries. In DSMS, queries are continuous in the sense that they are evaluated continuously as data items continue to arrive, generating results over time reflecting the data items seen so far. Three querying paradigms have been proposed [GÖ03a] by which the users can express their queries : relation-based, object-based and procedural.

Relation-based Languages. This type of languages uses an SQL-like syntax enhanced with support for sliding windows and ordering. CQL [AW04], developed to be used in the STREAM

DSMS [ABB+03] is an example of a relation-based language. The main modification made to standard SQL is a sliding window specification provided in the query's FROM clause. CQL is based on three classes of operators over streams and relations : relation-to-relation, stream-to-relation and relation-to-stream. In CQL a stream S is a possibly infinite bag of elements $\langle s, \tau \rangle$, where s is a tuple belonging to S and $\tau \in \mathcal{T}$ is the timestamp of the element. A relation R is a mapping from $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to R. A relation-to-relation operator corresponds to standard relational algebraic operators, it takes one or more relations $R_1, ..., R_n$ as input and generates a relation R as output. A stream-to-relation operator takes a stream S and generates a relation R with the same schema as S, this operator is based on the concept of sliding window. A time-based sliding window one stream S is specified by following the name of the stream with the RANGE keyword and a time interval enclosed in brackets. A relation-to-stream operator takes a relation R as input and generates a stream S as output. Conceptually, an unbounded stream is converted to a relation by way of sliding windows where the query is computed and the results are converted back to a stream. A window size in time units on a stream X contains a historical snapshot of a finite portion of the stream. An example query, computing a join of two timed-based windows of size 5 minutes each is shown below.

```
SELECT Distinct X.A
FROM X[RANGE 5 min], Y[RANGE 5 min]
WHERE X.B=Y.B
```

This query contains a stream-to-relation operator and a relation-to-relation operator that performs projection and duplicate elimination.

Object-based Languages. In object-based languages, streams are modeled as hierarchical data types. For instance, in the Tribeca [Sul96] network monitoring system the UDP/IP and TCP/IP types both inherit from the IP type. A query has a single source stream and at least one result stream. The query language of Tribeca supports the following operators over the entire input stream or over a sliding window : projection, selection, aggregation and join, multiplex and demultiplex (in order to support task of traffic analysis that partition a stream into a substreams, process the substreams, and then recombines the results of the substream analysis).

Procedural Languages. In this type of languages, the user specifies the data flow using the popular boxes and arrows paradigm found in most process flow and workflow systems. Borealis [AAB+05] is a DSMS that inherits from the SQuAl procedural language of the Aurora [ACc+03] DSMS. A query plan is constructed using a graphical interface by arranging boxes representing query operators joined by directed arcs representing the data flow between boxes. However, an optimization phase may re-arrange, add or remove operators. SQuAl has nine primitive operators (filter, map, union, aggregate, join, bsort, resample, read and update) and a set of system-level drop operators used to deal with system overload.

2.1.2 Query Operators

Conceptually, a query operator may be thought of as a function that consumes input streams, stores some state, performs computation when new data arrive, modifies the state and outputs results. DSMSs support standard relational operators such as join and aggregate, however their implementation is different. Regarding the join operator, a new implementation is justified by the following reasons. First, the join operator has a blocking behaviour because to produce the

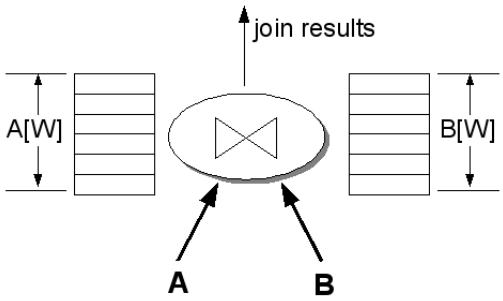


Figure 2.1 – A join operator with sliding windows

first result it must see its entire input. Since data streams may be infinite, a blocking operator will never see its entire input not being able to produce any result. However, this problem has been solved by replacing traditional join blocking operators with streaming symmetric join operators [VNB03] that processes tuples from the streams in an arbitrary interleaved fashion, i.e. for each arrival on one of the inputs, the state of the other input is probed to generate new results. Second, the join operator stores a state that grows linearly with the size of its inputs matching every tuple from one relation with every tuple from the other relation. Since data streams are potentially unbounded in size, its is not possible to store state continuously and match all tuples. To solve this problem, the join operator matches tuples considering a recent portion of the streams only. Generally this portion of the streams is based on a sliding window that explicitly defines the state of the operator as the set of tuples in the window. Usually two kinds of window constraints can be used over a join operator to limit the size of its states : tuple-based and time-based. For a tuple-based window, the window size is represented as the number of tuples, and for a time-based window the size is represented as a time frame. For example, two input streams (see Figure), A and B, both with tuples that contains a *time* attribute, and a window size W), the operator matches tuples that satisfy $|a.time - b.time| \leq W$.

In general a join operator, implementing sliding windows to limit the size of its states, executes a 3-step process referred to as the *purge-join-insert* algorithm [WRGB06]. For a newly arriving tuple $a \in A$: (1) a is used to purge tuples stored in window $B[W]$, (2) a is probed with tuples in $B[W]$ possibly producing join results, and (3) a is inserted in $A[W]$. Symmetric steps are executed for a B tuple.

In this section we present the three major join operators used for executing continuous multi-join queries, BJoin[], MJoin[VNB03] and Eddy[AH00]. In Bjoin a query plan is composed of binary join operators that store intermediate results, while an MJoin is an operator that takes symmetrically all the streams joining the arriving tuples with the remaining streams in a particular order without storing intermediate results. In the Eddy operator queries are processed without fixed plans. Instead, query execution is conceived as a process of routing tuples through operators where a tuple routing operator adjusts the routing order of tuples on a per-tuple basis.

2.1.2.1 The Binary Join Tree Operator (BJoin)

In a Binary Join Tree operator (BJoin) [VNB03] the query plan is composed of binary join operators. Each binary join operator applies a symmetric join algorithm that processes tuples from the streams in an arbitrary interleaved fashion based on the order of arrival. In BJoin, each

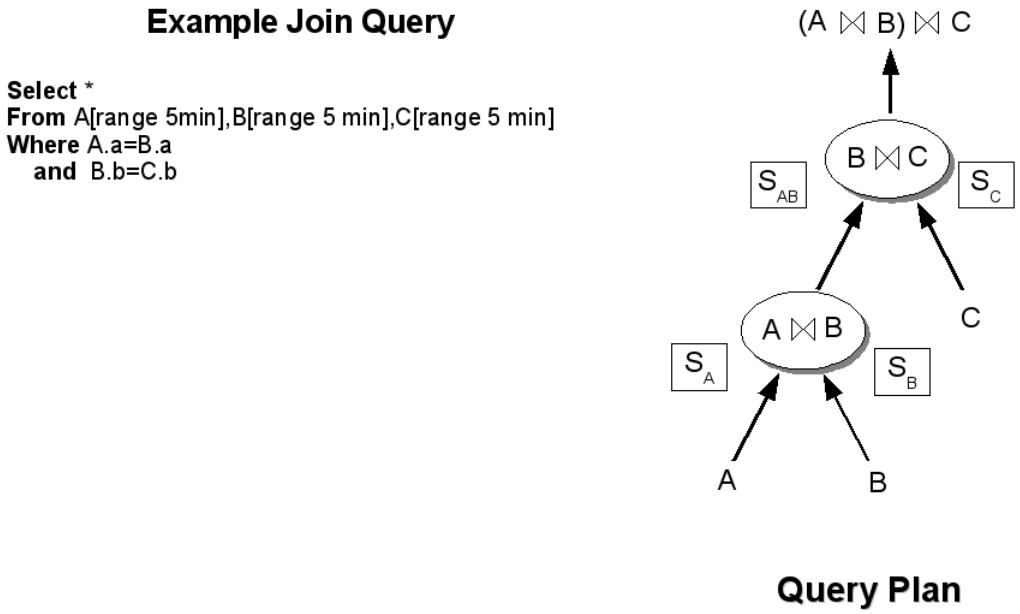


Figure 2.2 – A 3-way continuous join query using the BJoin operator

binary operator keeps two states that stores tuples that the operator has received so far. There are states that store the tuples received directly from a stream, such as state S_A in Figure 2.2, and others states, such as S_{AB} that store intermediate join results. To deal with infinite inputs, the states can be maintained using sliding windows and the join tuples are calculated using the *purge-probe-insert* algorithm described in Section 2.1.2. Consider, for example, the operator $A \bowtie B$ in Figure 2.2. An incoming tuple a_1 from input stream A first *purges* expired tuples of stream B stored in S_B. Then, it *probes* S_B to produce new join results involving tuple a_1. If new join results are produced, they are stored in the state S_{AB}. Finally, a_1 is *inserted* into S_A. The same process applies to any tuple from streams B and C. However, incoming tuples in C are probed directly with the intermediate results stored in S_{AB} instead of being probed with S_A and S_B separately avoiding the recomputation of intermediate results. Not maintaining such intermediate results can limit performance when incoming tuples in C arrive with the same join attribute value.

Existing optimization techniques such as the *min-state algorithm* [Zhu06] aim at minimizing the total number of intermediate results, thus reducing memory to store them as well as the processing costs (CPU) in generating future join results. The *min-state algorithm* is a greedy-based algorithm that computes a BJoin plan in polynomial time as follows. The input to the algorithm is a join graph $G = (V, E)$ that represents a multi-join query, where V represents the set of input streams, marked by its stream name V_i and its arrival rate λ_{V_i}, and an edge $(V_i, V_j) \in E$ represents a join predicate between two streams marked by the selectivity $\sigma_{V_iV_j}$ of the join $V_i \bowtie V_j$. Consider a 5-way join $A \bowtie B \bowtie C \bowtie D \bowtie E$ in Figure 2.3(a) with sliding windows of size $W = 1$. The algorithm ranks the edges using the following expression $\lambda_{V_i}\lambda_{V_j}\sigma_{V_iV_j}$. The smallest value is selected and a join is generated with its vertex. Thus, the edge (A, D) is selected first, is merged into a single vertex AD and the join $A \bowtie D$ is formed as shown in Figure 2.3(a). The arrival rate of the new AD vertex is the output rate of tuples generated

by $A \bowtie D$ that can be calculated as $\sigma_{AD}(\lambda_A(\lambda_D W_D) + \lambda_D(\lambda_A W_A)) = 0.1(5 \times 5 + 5 \times 5) = 5$. Once the join graph updated, the algorithm picks the following smallest weighted edge. The edge CE is chosen and the join $C \bowtie E$ is formed (see Figure 2.3(b)). The arrival rate of the new vertex CE is 8. The same procedure is repeated (see Figure 2.3(c)) and the join $B \bowtie C \bowtie E$ is generated. Finally, as shown in Figure 2.3(d) the output of the algorithm is a binary join tree that minimizes the size of intermediate states. The *min-state algorithm* does not guarantee to always find an optimal BJoin tree, thus leading the optimizer to be more conservative because it requires more resources than the query actually needs. However, it was chosen for its efficiency [Zhu06] i.e. a good plan is found quickly, which is much needed by continuous query processing.

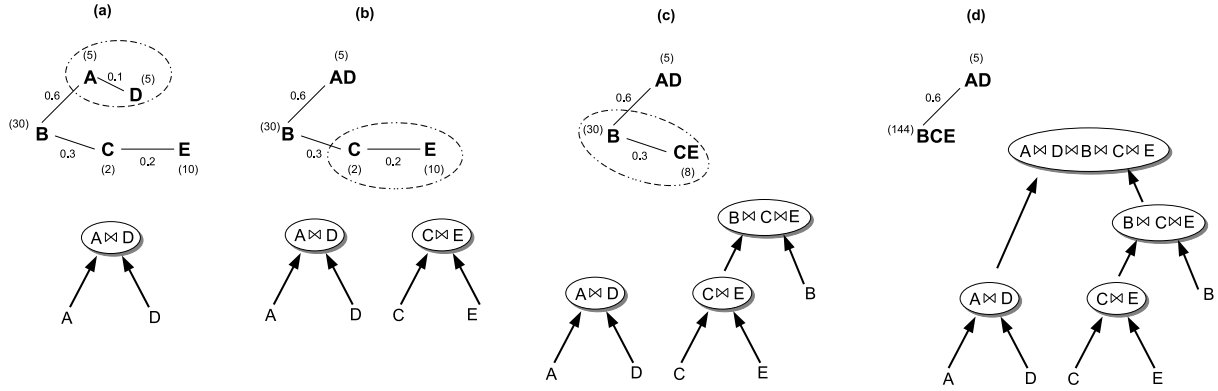


Figure 2.3 – Min-State Algorithm

2.1.2.2 The Multi-Way Join Operator

The basic idea of the Multi-Way Join operator (MJoin) [VNB03] is to generalize the symmetric binary hash join and the XJoin [UF00] to work for more than two inputs streams. Mjoin considers n inputs streams symmetrically and by allowing the tuples from the streams to arrive in an arbitrary interleaved fashion. The basic algorithm of MJoin creates as many hash tables (states) as there are join attributes in the query. When a new tuple from a stream arrives into the system, it is probed with the other $n-1$ streams in some order to find the matches for the tuple. The order in which the streams are probed is called the *probing sequence*. Note that depending of the query, not all the streams are always eligible to be probed. To this end, an MJoin implements a lightweight *tuple router* that routes the tuples to relations generating correct join probes.

Figure 4.3 shows an MJoin operator for a 3-way continuous join query expressed using CQL [AW04]. There are three hash tables corresponding to the three join attributes of the query. An MJoin operator is ready to accept a new tuple on any input stream at any time. Upon arrival, the new tuple is used to probe the remaining hash tables and generate a result as soon as possible. For example, the following steps are executed when a new tuple $a \in A$ arrives :

- a is inserted into the A hash table.
- The only valid *probing sequence* for an A tuple is $B \rightarrow C$ (see Figure 4.3). Note that an A tuple is not eligible to be probed into the C hash table directly, since they do not contain

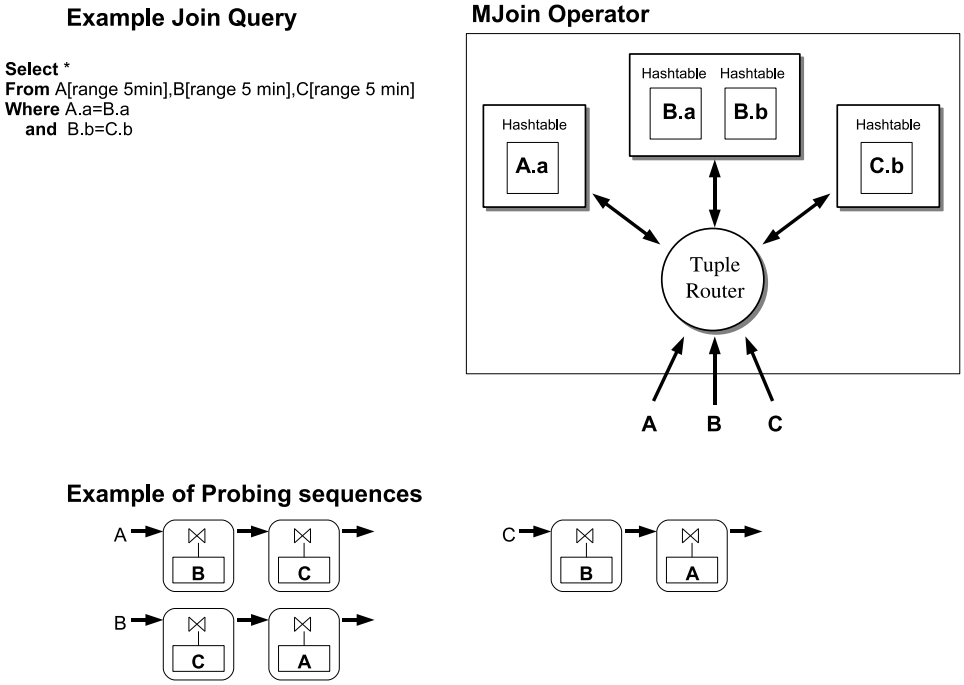


Figure 2.4 – A 3-way continuous join query using the MJoin operator

the same join attribute.

- a is used to probe the hash table on B to find tuples that satisfy the join predicate $A.a = B.a$. Intermediate tuples are generated by concatenating a with the matches $c_i \in C$, if any.
- If any result tuples were generated, they are routed to the C hash table in order to find tuples that satisfy the join predicate $B.b = C.b$.

Similarly, when a new C tuple arrives, it is inserted into the hash table on C. It is used to probe the hash table on B, since it is not eligible probing directly into the A hash table, and the intermediate tuples (if any) are routed to the A hash table to find tuples that satisfy the join predicate $A.a = B.a$. In the case of new B tuples, there are two probing sequences $C \rightarrow A$ and $A \rightarrow C$. Choosing a *probing sequence* is very important in MJoin because it must ensure that the smaller number of intermediate results is generated. This process can be supported by an adaptive algorithm [BMM+04].

Let us explain the execution of MJoin with the example query of 4.3 and the following tuple arrival $a_1, a_2, b_1, b_2, b_3, c_1, c_2, a_3, c_3$ (see Figure 2.5). We assume that the tuples arrive in the following way generating the following partitions $A_1 = \{a_1, a_2\}$, $B = \{b_1, b_2, b_3\}$, $C_1 = \{c_1, c_2\}$, $A_2 = \{a_3\}$, $C_2 = \{c_3\}$ (see Figure 4.3). This generates the following query plans :

- $(C_1 \bowtie B) \bowtie A_1$, C_1 tuples arrive last and the *probing sequence* $B \rightarrow A$ is chosen (see Figure 4.3).
- $(A_2 \bowtie B) \bowtie C_1$, when the tuple a_3 arrives the *probing sequence* $B \rightarrow C$ is chosen.
- $(C_2 \bowtie B) \bowtie (A_1 \cup A_2)$, when c_3 arrives the *probing sequence* $B \rightarrow A$ is chosen.

The execution of a MJoin operator can be seen as a sequence of query plans executed using left-deep pipeline plans where the internal state of hash tables is determined solely by the source

tuples that have arrived so far.

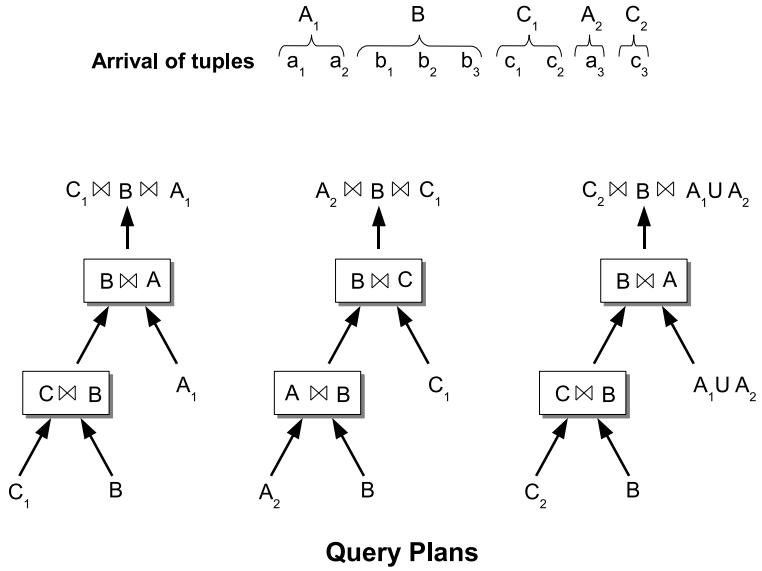


Figure 2.5 – MJoin and the left-deep pipeline plans generated by an arrival of tuples

MJoin is very attractive when processing continuous queries over data streams because the query plans can be changed by simply changing the *probing sequence*. Sliding windows are adopted to deal with infinite inputs limiting the size of hash tables and the *purge-probe-insert* algorithm (see Section 2.1.2) is used to calculate join tuples.

The principal drawback of MJoin is the recomputation of intermediate results because intermediate tuples generated during query execution are not stored for future use. In Figure 2.5, the first query plan generates $(C_1 \bowtie B)$ intermediate results which are not stored and could be used in the query plan $(A_2 \bowtie B) \bowtie C_1$. A solution to this problem is proposed in [VNB03] where a MJoin is broken into smaller MJoins or into a tree of binary joins. Another solution, called adaptive caching [BMWM05], is to add/remove temporary caches of intermediate result tuples.

Caching intermediate results. The *A-caching* algorithm [BMWM05] improves the performance of MJoin using caches to store intermediate result tuples. In *A-caching* the performance of a continuous join depends on the *probing sequence* and caching. This approach follows a two-step process, to improve the performance of MJoin :

- A *probing sequence* is chosen, independently for each stream, using the *A-Greedy* algorithm [BMM+04].
- For a given *probing sequence*, *A-caching* may decide to add a cache in the middle of the pipeline.

A intermediate result cache $\mathbb{C}_O^{\mathbb{S}}$ is an associative store, where $\mathbb{S}$ is the stream for which it is maintained and O is the set of operators belonging to the *probing sequence* for which it stores the intermediate result tuples. The cache entries contain key-value pairs (u, v) where u is the value of the join attribute and v are the result tuples generated by probing into O. A cache

supports the following operations :

- $create(u, v)$ adds the key-value pair into the cache.
- $probe(u)$ returns a *hit* with value v if $(u, v) \in \mathbb{C}_O^{\mathbb{S}}$, and a *miss* otherwise.
- $insert(u, r)$ adds r to set v.
- $delete(u, r)$ removes r from set v for determined attribute values.

Figure 2.6 shows a $B \bowtie C$ cache for the query and the *probing sequences* presented in Figure 4.3. When an A tuple arrives, the cache is consulted first to verify if there are results already cached. If so, the *probing sequence* can be avoided, thus saving the work that would otherwise has been performed to generate the result tuples. If there is a *miss*, the *probing sequence* continues normally and inserts back the computed result into the cache. When new B and C tuples arrive, the cache must be updated if the arriving tuples generate $B \bowtie C$ or $C \bowtie B$ intermediate results. In the same way, when tuples of A and B are dropped out of their respective sliding windows the cache must be updated. This latter operation could be computationally expensive, making important the choice of which caches to maintain.

In *A-caching*, giving a *probing sequence* the caches are selected adaptively to optimize the performance based on profiled costs and benefits of caches. Therefore, caches can be added, populated incrementally and dropped with little overhead, without compromising the join results.

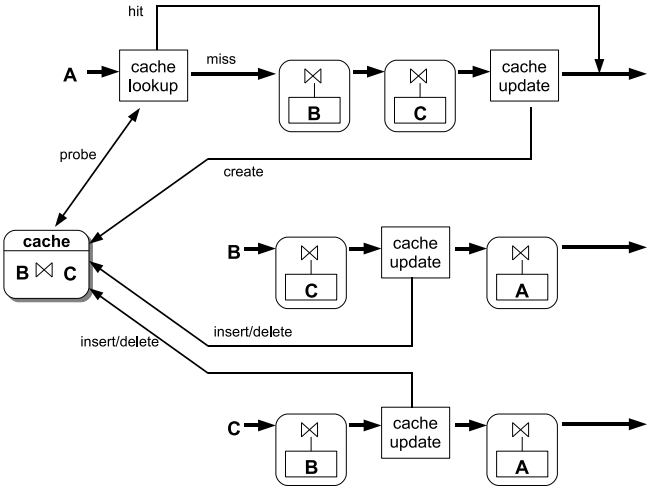


Figure 2.6 – Query plans with a $B \bowtie C$ cache

2.1.2.3 The Eddy Operator

The Eddy operator is designed to enable fine-grained runtime control and adaptively approach the optimal order of join operations at runtime [AH00]. Queries are executed without fixed plans. Instead, an optimized join order for each incoming tuple is computed individually. An Eddy is a tuple routing operator between data sources and query operators as joins. It monitors the execution and makes routing decisions by sending tuples from the data sources through query operators. As a result, the routing destinations for tuples alone determine the query plans executed by the Eddy.

Figure 2.7 shows an Eddy operator for a 3-way continuous join query expressed using CQL [AW04]. Along with the Eddy operator, two symmetric binary hash join operators are instantiated. Tuples from A, B and C are routed through $A \bowtie B$ and $B \bowtie C$. Since an operator expects tuples of certain sources, arbitrary routing of tuples is not allowed. For instance, tuples from A should not be routed to $B \bowtie C$. In the query of Figure 2.7, the valid routing options for different types of tuples (shown on the data flow edges) are as follows :

- A and C tuples can only be routed to the $A \bowtie B$ and $B \bowtie C$ operator respectively.
- B tuples can be routed to either of the two join operators.
- Intermediate AB and BC tuples can only be routed to $B \bowtie C$ and $A \bowtie B$ operator respectively.
- ABC result tuples are routed to the output.

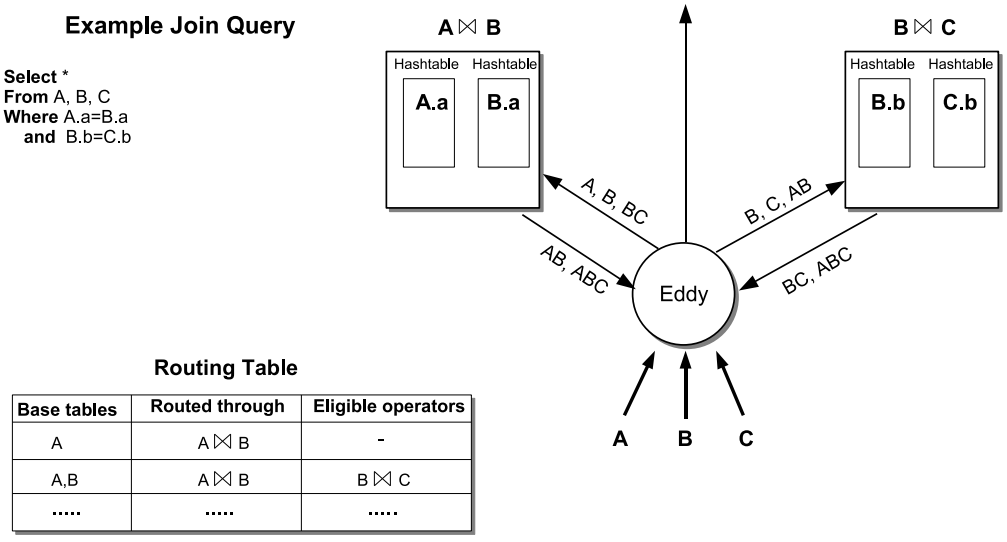


Figure 2.7 – A 3-way join query using the Eddy operator

The validity of routing decisions is supported by using a *tuple signature* composed of the set of base relations that a tuple contains and the operators it has already been routed through. A high level representation of a *tuple signature* is shown in Figure 2.7, valid routing destinations can be added configuring a routing table. For efficient storage and lookups, a compact representation of the *tuple signature* is proposed. To this end, each tuple is assigned a *ready* and a *done* bitset that respectively represent the operators that are eligible to process the tuple, and the operators that the tuple has already been through. If all the *done* bits are on, the tuple is sent to the eddy's output ; otherwise it is sent to another eligible operator. A join ordering is expressed turning on only the *ready* bit corresponding to the operator that can be executed initially. When an operator returns a tuple to the eddy, it turns on the *ready* bit of the following operator in the join ordering. A latter implementation of Eddies in PostgreSQL [Des04] encodes the bitmaps as integers to index a routing table initialized at the beginning of query execution. Thus, valid destinations for tuples with the same signature are found efficiently. However, the size of the routing table is exponential in the number of operators and hence this approach is not suitable

for queries with a large number of operators.

An Eddy has a tuple routing scheme that monitors the behaviour of the operators (cost and selectivity) and accordingly routes tuples through the operators. Two tuple routing schemes are proposed in [AH00], *back-pressure* and *lottery scheduling*. The *back-pressure* works as follows : the processing of a tuple is more slow in a high cost operator than in that of low cost. This generates larger input queue sizes for high costs operators. If the length of input queues is fixed, the Eddy operator is forced to route tuples to an operator of lower cost before routing to those of higher cost. Under the *lottery scheduling*, each time a tuple is routed to an operator, it obtains a ticket. When the operator returns a tuple to the Eddy, one ticket is debited. Thus, the number of tickets is used to roughly estimate the selectivity of an operator. When two operators are eligible to process a tuple, the operator with more tickets has higher probability to process it. By doing this, the Eddy is very adaptive and the join order can be changed at runtime.

Intelligent routing decisions are the key to achieve good performance. However, it is difficult to predict how the decisions made by the Eddy will affect the execution of subsequent tuples. The problem is the state accumulated inside the join operators. A tuple stored in the state of a join operator can effectively determine the order of execution for subsequently arriving tuples from other tables even after the Eddy has switched the routing policy. As a result, the Eddy effectively continues to emulate an suboptimal plan. To illustrate this point, consider the query of Figure 2.7. At the beginning the Eddy emulates the plan $(A \bowtie B) \bowtie C$ knowing that the data source of A is stalled, which makes the $A \bowtie B$ operator an attractive destination for routing B tuples. Some time later, a great quantity of A tuples arrive and it becomes apparent that the plan $A \bowtie (B \bowtie C)$ is the better choice. The Eddy switches the routing policy so that subsequent B tuples are routed to $B \bowtie C$ first. Unfortunately the Eddy continues to emulate the suboptimal plan $(A \bowtie B) \bowtie C$, even if the routing policy for B tuples has changed, because of all the previously seen B tuples are still stored in the internal state of the $A \bowtie B$ operator. As A tuples arrive, they must join with these B tuples before the B tuples are joined with C tuples. To tackle this problem SteMs [RDH03] and STAIRs [DH04] are proposed.

The State Modules (SteMs) architecture is an extension of the Eddy architecture that ensures that the state stored in the join operators is entirely independent of routing history. To this end, SteMs does not store intermediate results. The main operator is a SteM, which is instantiated for each attribute of each base relation addressed in the join predicates (see Figure 2.8). The query is executed by routing tuples through these operators. Tuples arriving from each base relation are first built into their own SteM and then used to probe the other relations' SteMs to get the join results. Considering the query of Figure 2.8, when a new A tuple arrives, it is :

1. inserted into the A SteM.
2. probed against B tuples stored in the B SteM to find matching tuples corresponding to $A \bowtie B$.
3. the resulting AB tuples are probed against the C tuples stored in the C SteM in order to generate ABC results. Intermediate AB tuples are not stored anywhere. Thus, the state accumulated into SteMs is independent of the routing history.

However, this solution has two significant drawbacks :

- **Re-computation of intermediate results** : since intermediate results are not stored anywhere, they are re-computed each time they are needed.
- **Constrained plan choices** : query plans that can be executed for any new tuples are constrained because even if the Eddy knows the existence of an optimal query plan, this

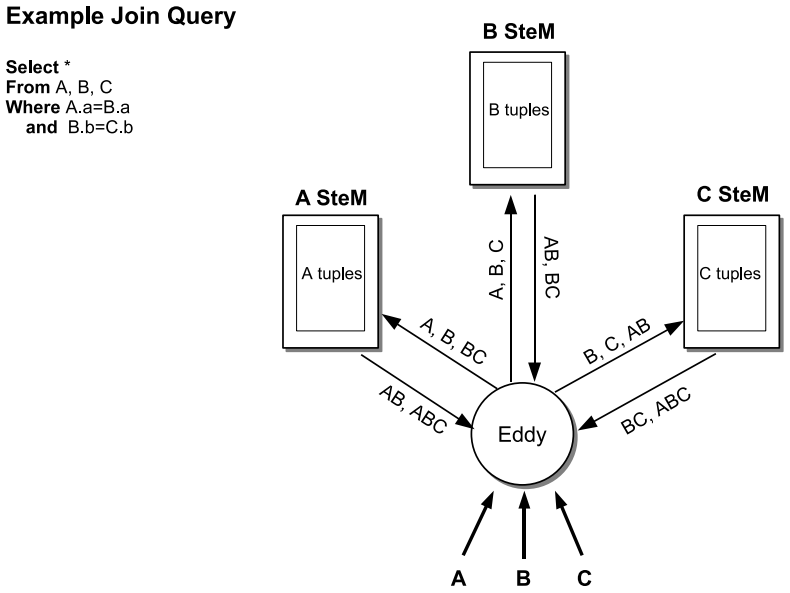


Figure 2.8 – SteMs for a 3-way join query

plan is not feasible. For example, any new A tuple must join with B tuples (stored in the SteM on B) first and then with C tuples (stored in the SteM on C). This restricts the query plan for new A tuples to be $(a \bowtie B) \bowtie C$. A new optimal query plan such as $a \bowtie (B \bowtie C)$ cannot be proposed because the absence of $B \bowtie C$ tuples.

The STAIR operator exposes the state stored in the operators and allows to the Eddy manipulate this state in order to reverse any bad routing decisions. A STAIR on relation A and attribute a, denote by $A.a$, contains either tuples from A or intermediate tuples that contain a tuples from A, and supports the following two basic operations :

- $insert(A.a, t)$: store a tuple t, that contains a tuple from A, inside the STAIR.
- $probe(A.a, val)$: given a value val from the $A.a$'s domain, return all the matching tuples stored in the STAIR $A.a$.

Figure 2.9(a) shows how STAIRs are instantiated. Each join operator is replaced with two STAIRs that interact with the Eddy directly. These STAIRs are called duals of each other. The query execution using STAIRs is similar to query execution using an MJoin operator in the following sense : when a new tuple arrives, the Eddy performs an *insert* on one STAIR, and a *probe* into its dual. This is a property, called *Dual Routing Property*, always obeyed during query execution.

The Eddy manipulates the state inside the STAIRs using two primitives that provide the ability to reverse any bad routing decisions made in past.

- $Demotion(A.a, t, t')$: this operation reduces an intermediate tuple t stored in the STAIR $A.a$ to a sub-tuple t' of that tuple. Intuitively, this operation undoes a tuple that was done earlier during execution.
- $Promotion(A.a, t, B.b)$: this operation replaces a tuple t with super tuples of that tuple generated using another join in the query. Intuitively, this operation reroutes the tuple t with to the new join ordering. To promote a tuple, the following steps are performed :

1. Remove t from $A.a$.
2. Insert t into $B.b$.
3. Probe the dual of $B.b$ using the tuple t.
4. Insert the resulting super-tuples of t (if any) back into $A.a$.

This process of moving state (tuples) from one STAIR to another is referred to as *state migration*. As an example, Figure 2.9(a) shows state maintained inside the joins operators at time τ for the query example. At time τ the system has received the tuples a_1, b_1, b_2, b_3, c_1. Tuples b_1 and b_2 have been routed to $A \bowtie B$ and b_3 is routed to $B \bowtie C$. At this time the Eddy knows that routing b_1 and b_2 to $A \bowtie B$ was a mistake. Thus, this prior routing decision can be reversed using *Demotion* and *Promotion* operations. Thus, tuples a_1b1 and a_1b_2 are replaced by the sub-tuples $b1$ and $b2$ after the call to *Demotion* (see Figure 2.9(b)). In general, this operation can be interpreted as Demote all the $A \bowtie B^A$ tuples in $B.b$ to B^A, where B^A are the tuples of B routed to $A \bowtie B$. Figure 2.9(c), we see the Eddy after the call to *Promotion*. Tuples routed erroneously to $A \bowtie B$ are replaced with super-tuples of these tuples that are generated when the routing decision is . Thus, b_1 and b_2 are rerouted to $B \bowtie C$ and result tuples are inserted back into STAIR $B.a$. As a result, bad routing decisions are reversed and the Eddy keeps the ability to adapt over the time.

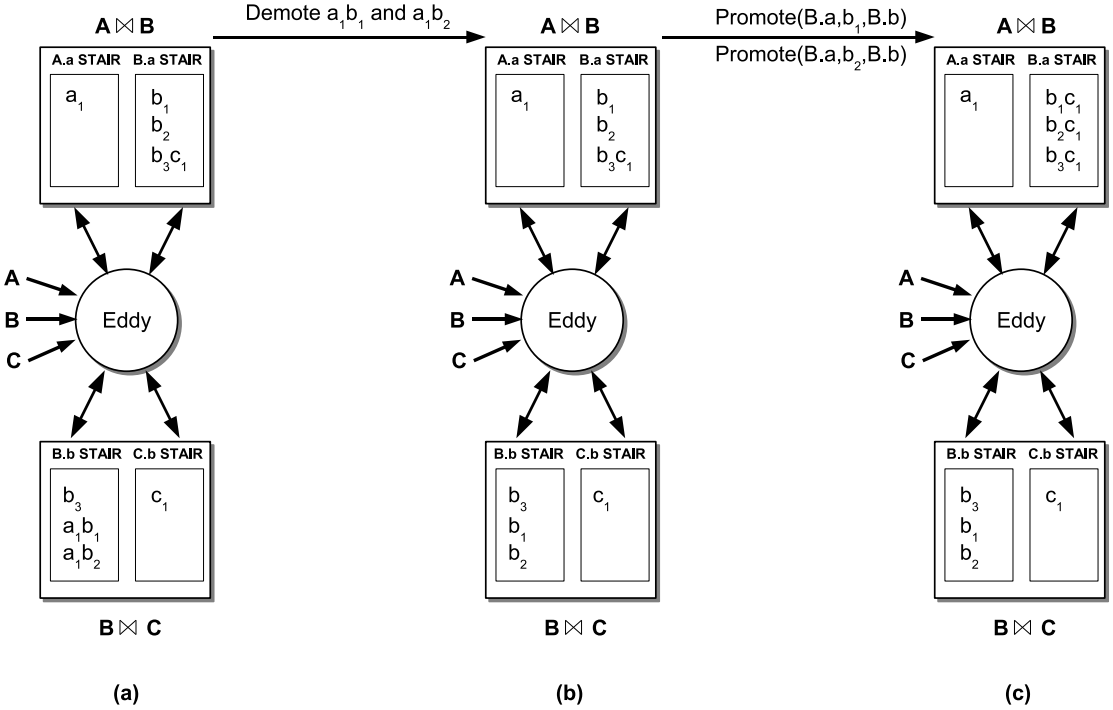


Figure 2.9 – An example of state migration using STAIRs

2.1.3 Data Stream Management Systems

We now survey some of the main DSMSs in approximate chronological order of when they started. We present their primary contributions and how continuous queries are processed including query languages and operators.

TelegraphCQ [CCD+03] is one the firsts to process continuous queries focusing in adaptive query processing. The software architecture of TelegraphCQ is based on an adaptation of PostgreSQL to enable shared processing of continuous queries over stream sources. Tuples are read from external sources using user-defined data acquisition functions called *wrappers*. A *wrapper*, declared and managed using the PostgreSQL user-defined function infrastructure, processes data read from a networked source and generates tuples in the appropriate format. Queries are expressed using the SQL Select statement, including a Window clause to specify the window size of streams operations, and are executed without fixed plans. Instead, an optimized join order for each incoming tuple is computed individually using an Eddy operator (see Section 2.1.2.3).

STREAM [ABB+03] is a DSMS built from scratch. Many aspects of stream processing as query languages, resource management and statistics computation are explored during the development of STREAM. Queries are specified in CQL [AW04] and once a query arrives to the system a query plan is compiled from it. A query plan is composed of a tree of operators (which perform the processing) connected by queues with synopses (which store operator state) attached to operators as needed. Regarding join queries, they are executed using an MJoin operator [VNB03] optimized by using the *A-caching* algorithm [BMWM05] that places and removes caches of intermediate result tuples (see Section 2.1.2.2).

Borealis [AAB+05] is a distributed DSMS which inherits the core data model and stream processing functionality from Aurora [ACc+03]. The design of Borealis focuses on fault tolerance and overload management. Queries are specified using query diagrams which can be seen as a network of query operators whose processing is distributed to multiple nodes. Fault tolerance is tackled using a replication-based approach [BBMS08] which maintains consistency (users receive correct results) and low processing latency. Overload is tackled dropping tuples [TcZ07] using a load shedding plan composed of a set of drop operators placed at specific arcs of the query diagram. Considering that the system cannot spend large amount of time calculating the best plan at runtime, an off-line algorithm builds a set of plans in advanced which can be invoked under different combinations of input load. Moreover, under dynamic environment a distributed algorithm that efficiently computes changes to plans is proposed.

DCAPE [Zhu06] is a distributed DSMS focusing in adaptive query processing. The architecture of DCAPE was conceived to provide adaptive services at all levels of query processing including dynamic query optimization, dynamic plan migration and distributed adaptation. Unlike TelegraphCQ fine-grained level of adaptivity, DCAPE works with pre-defined optimized query plans re-optimized using adaptive techniques. Queries are executed using partitioned parallelism and re-optimized when the query plan becomes inefficient due to changes of the data characteristics (selectivity and stream rate). The change of query plan is performed online without affecting the correctness of the output. This transition is referred as dynamic plan migration. The heterogeneity of plan shapes among different machines due to local query optimizations is integrated with distributed load balancing [ZR07] using protocols that can balance the load between machines handling the complexity caused by local plan changes. This process is referred as distributed adaptation.

Table 2.1 – Summary of join operators, their primary contributions, drawbacks and improvements.

Operator	Primary contribution	Drawback	Improvement
BJoin	First approach to replace blocking operators with streaming symmetric operators.	The growth of intermediate results increase the processing cost in generating join results.	A greedy algorithm that reduces the total number of intermediate results.
MJoin	High adaptivity obtained using different query plans for each stream.	Recomputation of intermediate results.	Adaptive caching to reuse intermediate tuples produced during execution.
Eddy	High adaptivity obtained calculating an optimized join order for each incoming tuple.	Execution of suboptimal plan even if the routing policy has changed	There is two approaches to improve Eddy performance : 1) Use SteMs that do not store intermediate results and 2) manipulate state store in join operators using STAIRs in order to reverse bad routing decisions.

2.1.4 Conclusion

In this chapter, we discussed query languages, join query operators and how the different DSMSs process continuous queries. In declarative languages as CQL [AW04] the sintax is similar to SQL, but their semantic is considerably different. Object based languages are conceived principally for specific domain applications as network monitoring [Sul96]. In procedural languages [ACc+03] the user specifies how the data flows through the system allowing, as in the case of Borealis DSMS, the graphical specification of a drop operator and how it is deployed over a set of computing nodes.

We presented three join operators, BJoin [VNB03], MJoin [VNB03] and Eddy [AH00]. Table 2.1 summarizes the primary contribution of join operators, their drawbacks and improvements proposed. BJoin is the natural extension of a classical binary tree of join operators to the stream context. Intermediate join results are stored for future use and queries are executed using a fixed query plan. Mjoin provides high adaptivity using different *probing sequences* for each stream. Thus, depending of the arriving tuple a different left-deep query plan defined by the *probing sequence* is executed. However, since MJoin does not store intermediate join results, the recomputing of intermediate results can generate high processing costs. To tackle this problem, a solution based on adaptive caches of intermediate results is proposed. Basically, an adaptive cache can be instantiated, populated and dropped (if necessary) without compromising of join results. The Eddy operator continuously reoptimizes a query making per-tuple routing decisions. Thus, an optimal query plan is calculated for each incoming tuple. This provides high adaptivity, but making per-tuple routing decisions may suggest high processing costs. However, an study of overhead of Eddies [Des04] shows that its benefits can be obtained with some minor tuning.

We presented different DSMSs, Table 2.2 summarizes these DSMSs and their primary contributions. Each DSMS has been developed emphasizing a particular research topic of stream

Table 2.2 – Summary of DSMSs and their primary contributions.

DSMS	Primary contribution
TelegraphCQ	Operators for adaptive query processing.
STREAM	Adaptive caching for continuous queries and query language.
Borealis	Techniques for fault-tolerance and load management.
DCAPE	Integrates local query optimization and distributed load balancing

processing. TelegrapCQ emphasizes adaptive query processing developing the highly adaptive Eddy operator, in the others DSMSs the choice of operators is not very important since their focus is in other aspects of stream processing as resource management [ABB+03], fault-tolerance [AAB+05] and load balancing [Zhu06].

2.2 Query Processing in P2P networks

P2P systems adopt a completely decentralized approach to resource management. By distributing data storage, processing, and bandwidth across autonomous peers in the network, they can scale without the need for powerful servers. P2P systems have been successfully used for sharing computation, e.g. Setihome [ACK+02] and Genomehome [LSSP03], communication, e.g. ICQ and Jabber, internet services, e.g. P2P multicast systems [CJK+03] and security applications [KMR02][VATS04], or data, e.g. Gnutella, KaZaA and PIER [HHL+03].

There are several features that distinguish P2P systems from traditional distributed database systems (DDBS) and make it difficult to provide advanced data management services over P2P networks [NOTZ03] : Peers are very dynamic and can join and leave the system anytime. However, in DDBS, nodes are added to and removed from the system in a controlled manner. Usually there is no predefined global schema for describing the data which are shared by the peers. In P2P systems, the answers to queries are typically incomplete. The reason is that some peers may be absent at query execution time. In P2P systems, there is no centralized catalog that can be used to determine the peers that hold relevant data to a query. However, such a catalog is an essential component of DDBS.

Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems. This work leads to structured solutions based on distributed hash tables (DHT), e.g. CAN [RFH+01] and Chord [SMK+01], or hybrid solutions with super-peers [NSS03]. Although very useful, most of the initial P2P systems are quite simple (e.g. file sharing), support limited functions (e.g. keyword search) and use simple techniques (e.g. resource location by flooding) which have performance problems. In order to overcome these limitations, recent works have concentrated on supporting advanced applications which must deal with semantically rich data (e.g. XML documents, relational tables, etc.) using a high-level SQL-like query language, e.g. ActiveXML [ABC+03], Edutella [NWQ+02a], Piazza [TIM+03], PIER [HHL+03].

One of the main services which are needed for supporting advanced P2P applications is a query processing service which deals with schema-based queries. However, providing such a service in P2P systems is quite challenging because of the specific features of these systems, e.g. lack of a global schema. Most techniques designed for distributed database systems which statically exploit schema and network information no longer apply. New techniques are needed which should be decentralized, dynamic and self-adaptive. Therefore, novel techniques have been

proposed to perform decentralized schema mapping, to route queries to relevant peers without relying on a centralized catalog, and to execute queries, especially complex queries such as join queries, in a fully distributed fashion while taking into account the dynamic behavior of peers.

All P2P systems rely on a P2P network to operate. This network is built on top of the physical network (typically the Internet), and thus referred to as overlay network. The degree of centralization and the topology of the overlay network strongly impact the nonfunctional properties of the P2P system, such as fault-tolerance, self-maintainability, performance, scalability, and security.

The remainder of this chapter surveys query routing in P2P networks (Section 2.2.1), and different approaches implementing complex query facilities in DHT-based structured P2P networks (Section 2.2.3).

2.2.1 Query Routing in P2P Networks

The main problem for query processing in P2P systems is how to route the query to relevant peers, i.e. those that hold some data related to the query, [LW06]. Once the query is routed to relevant peers, it is executed at those peers and the answers are returned to the query originator. In this section, we describe the approaches for query routing in unstructured, DHT-based structured, and super-peer P2P systems.

2.2.1.1 Unstructured

In unstructured P2P networks, the overlay network is created in a nondeterministic (ad hoc) manner and data placement is completely unrelated to the overlay topology. Each peer knows its neighbors, but does not know the resources they have. Query routing is typically done by flooding the query to the peers that are in limited hop distance from the query originator. There is no restriction on the manner to describe the desired data (query expressiveness), i.e. key look-up, SQL-like query, and other approaches can be used. Fault-tolerance is very high since all peers provide equal functionality and are able to replicate data. In addition, each peer is autonomous to decide which data to store. However, the main problems of unstructured networks are scalability and incompleteness of query results. Query routing mechanisms based on flooding usually do not scale up to a large number of peers because of the huge amount of load which they incur on the network. Also, the incompleteness of the results can be high since some peers containing relevant data may not be reached because they are too far away from the query originator. Examples of P2P systems supported by unstructured networks include Gnutella, KaZaA and FreeHaven.

The approaches used in unstructured systems for query routing can be classified in the following groups [TR03a] : Breath-First Search (BFS), Iterative deepening, random walks, adaptive probabilistic search, local indices, bloom filter based indices, and distributed resource location protocol.

BFS. This approach, which is used originally by Gnutella for data discovery, floods the query to all accessible peers within a TTL (Time To Live) hop distance as follows. Whenever a query with a TTL is issued at a peer, called query originator, it is forwarded to all its neighbors. Each peer, which receives the query, decreases the TTL by one and if it is greater than one sends the query and TTL to its neighbors. By continuing this procedure, all accessible peers whose hop distance from the query originator is less than or equal to TTL receive the query. Each peer that receives the query executes it locally and returns the answers directly to the query originator

Iterative deepening. This approach [YGM02] is used when the user is satisfied by only one (or a small number) of the closest answers. In this algorithm, the query originator performs consecutive BFS searches such that the first BFS has a low TTL, e.g. 1, and each new BFS uses a TTL greater than the previous one. The algorithm ends when the required number of answers is found or a BFS with the predefined maximum TTL is done. For the cases where a sufficient number of answers are available at the peers that are close to the query originator, this algorithm achieves good performance gains compared to the standard BFS. In other cases, its overhead and response time may be much higher than the standard BFS.

Random Walks. In Random Walks [LCC+02], for each query, the query originator forwards k query messages to k of its randomly chosen neighbors. Each of these messages follows its own path, having intermediate peers forward it to a randomly chosen neighbor at each step. These messages are known as walkers. When the TTL of a walker reaches zero, it is discarded. Each walker periodically contacts the query originator, asking whether the termination condition is held or not. If the response is positive, the walker terminates. The main advantage of the Random Walks algorithm is that it produces $k \times TTL$ routing messages in the worst case (k=the number of walkers), a number which does not depend on the underlying network. The main disadvantage of this algorithm is its highly variable performance, because success rates and the number of found answers vary greatly depending on network topology and the random choices. Another drawback of this method is that it cannot learn anything from its previous successes or failures.

Adaptive probabilistic search. In Adaptive Probabilistic Search (APS) [TR03b], for each recently requested data, the peers maintain the data identifier and probability of returning the data by each of their neighbors. Given a query asking for a data, the query originator establishes k independent walkers and sends them to its neighbors. Each intermediate peer, which receives a walker, sends it to the neighbor that has the highest probability to return the requested data. Initially equal for all neighbors, the probability values are updated using either an optimistic approach or a pessimistic approach [TR03b]. APS has very good performance as it is bandwidth-efficient : the number of routing messages produced by it is very close to that of Random Walks. In spite of this, the probability of finding the requested data by APS is much higher than that of Random Walks. However, if the topology of the P2P system changes quickly, the ability of APS to answer queries reduces significantly.

Local indices. In this approach [YGM02], each peer p indexes the data shared by all peers which are within a certain radius r, i.e. the peers whose hop-distance from p is less than or equal to r. The query routing is done in a BFS-like way, except that the query is processed only at the peers that are at certain hop distances from the query originator. To minimize the overhead, the hop distance between two consecutive peers that process the query must be $2r + 1$. This allows querying all data without any overlap. The processing time of this approach is less than that of standard BFS because only a certain number of peers process the query. However, the number of routing messages is comparable to that of standard BFS. In addition, whenever a peer joins/leaves the network or updates its shared data, a flooding with $TTL = r$ is needed in order to update the peers' indices, so the overhead becomes very significant for highly dynamic environments.

Bloom filter based indices. In [RK02], the indexing of data is done using Bloom filters. Each peer holds d bloom filters for each neighbor, such that the i-th filter summarizes the data that can be found i hops away through that specific neighbor. When a peer receives a query requesting a data, it checks its local data, if the data is found it is returned to the query originator. Otherwise, the peer forwards the query to the neighbor who has the minimum numbered filter

that represents the data among its members. The advantage of representing the indexed data by Bloom filters [Blo70] is that they are space efficient, i.e. with a small space, one can index a large number of data. However, it is not possible to remove a data from a Bloom filter, so they are not easily adaptable to highly dynamic environments. In addition, it is possible that the Bloom filter wrongly returns a positive answer in response to a question asking the membership of a data item.

Distributed resource location protocol. In Distributed Resource Location Protocol (DRLP) [MK02], the peers index the location of all data which are the answer for recently issued queries. The indexing is done gradually as follows. Peers with no information about the location of a requested data, forward the query to a set of randomly chosen neighbors. If the data is found at some peer, a message is sent over the reverse path to the query originator, storing the data location at those peers. In subsequent requests, peers with indexed location information forward the query directly to the relevant peers. This algorithm initially spends many routing messages for query processing. In subsequent requests, it might take only one message to discover it. Thus, if a query is issued frequently, this approach is very efficient.

2.2.1.2 Structured

Structured networks have emerged to solve the scalability problem of unstructured networks. They achieve this goal by tightly controlling the overlay topology and data placement. Data (or pointers to them) are placed at precisely specified locations and mappings between data and their locations (e.g. a file identifier is mapped to a peer address) are provided in the form of a distributed routing table. Distributed hash table (DHT) is the main representative of structured P2P networks. A DHT provides a hash table interface with primitives $put(key, value)$ and $get(key)$, where key is an object identifier, and each peer is responsible for storing the values (object contents) corresponding to a certain range of keys. Each peer also knows a certain number of other peers, called neighbors, and holds a routing table that associates its neighbors' identifiers to the corresponding addresses. Most DHT data access operations consist of a lookup, for finding the address of the peer p that holds the requested data, followed by direct communication with p. In the lookup step, several hops may be performed according to nodes' neighborhoods.

Queries can be efficiently routed since the routing scheme allows one to find a peer responsible for a key in $O(log\ n)$ routing hops, where n is the number of peers in the network. Because a peer is responsible for storing the values corresponding to its range of keys, autonomy is limited. Furthermore, DHT queries are typically limited to exact match keyword search. Active research is ongoing to extend the DHT capabilities to deal with more complex queries such as range queries [RRHS04][JOV05] and join queries [HHL+03][ILK08].

Examples of P2P systems supported by structured networks include Chord [SMK+01], CAN [RFH+01], Tapestry [ZHS+04] and Pastry [RD01]. They use a chain mode propagation approach, where each node makes a local decision about to which node to send the request message next. The way by which a DHT routes the keys to their responsible depends on the DHT's routing geometry [GGG+03], i.e. the topology which is used by the DHT for arranging peers and routing queries over them. The routing geometries in DHTs can be [GGG+03] : tree, hypercube, butterfly, XOR and hybrid. In the following, we describe these geometries and discuss query routing.

Tree. Tree is the first geometry which is used for organizing the peers of a DHT and routing queries among them. In this approach, peer identifiers constitute the leaves of a binary tree of depth log n where n is the number of nodes of the tree. The responsible for a given key is the

peer whose identifier has the highest number of prefix bits which are common with the key. Let $h(p, q)$ be the height of the smallest common sub-tree between two peers p and q. For each $1 \leq i \leq log\ n$, each peer p knows the address of a peer q such that $h(p, q) = i$. This means that, for each $1 \leq i \leq log\ n$, the peer p knows a peer q such that the number of common prefix bits in the identifiers of p and q is i. The routing of a key proceeds by doing a longest prefix match at each intermediate peer until reaching to the peer which has the most common prefix bit with the key. The basic routing algorithms in Tapestry [ZHS+04] is rather similar to this algorithm. The tree geometry gives a great deal of freedom to peers in choosing their neighbors : each peer has $2i - 1$ options in choosing a neighbor in a sub-tree with height i. Thus, in total, each peer has about $2^{log\ n(log\ n-1)/2}$ options to select all its neighbors. Therefore, the tree geometry has good neighbor selection flexibility. However, it has no flexibility for message routing : there is only one neighbor which the message must be forwarded to, i.e. this is the neighbor that has the most common prefix bits with the given key.

Hypercube. The hypercube geometry is based on partitioning a d-dimensional space into a set of separate zones and attributing each zone to one peer. Peers have unique identifiers with $log\ n$ bits, where n is the total number of peers of the hypercube. Each peer p has $log\ n$ neighbors such that the identifier of the i-th neighbor and p differ only in the i-th bit. Thus, there is only one different bit between the identifier of p and each of its neighbors. The distance between two peers is the number of bits on which their identifiers differ. Query routing proceeds by greedily forwarding the given key via intermediate peers to the peer that has minimum bit difference with the key. Thus, it is somehow similar to routing on the tree. The difference is that the hypercube allows bit differences to be reduced in any order while with the tree, bit differences have to be reduced in strictly left-to-right order. The number of options for selecting a route between two peers with k bit differences is $(log\ n)(log\ n - 1)\ldots(log\ n - k)$, i.e. the first peer on the route has $log\ n$ choices, and each next peer on the route has one choice less than its predecessor. Thus, in the hypercube, there is great flexibility for route selection. However, for selecting its neighbors, a peer has only one choice. Thus, the hypercube geometry has no flexibility in the neighbor selection. The routing geometry used in CAN [RFH+01] resembles a hypercube geometry. CAN uses a d-dimensional coordinate space which is partitioned into n zones and each zone is occupied by one peer. When $d = log\ n$, the neighbor sets in CAN are similar to those of a $log\ n$ dimensional hypercube.

Ring. The Ring geometry is based on a one dimensional cyclic space such that the peers are ordered on the circle clockwise with respect to their identifiers. Chord [SMK+01] is a DHT protocol that relies on this geometry for query routing. In Chord, each peer has an m-bit identifier, and the responsible for a key is the first peer whose identifier is equal or follows k. Each peer p maintains the address of $logn$ other peers on the ring such that the i-th neighbor is the peer whose distance from p clockwise in the circle is $(2i - 1)\ mod\ n$. Hence, any peer can route a given key to its responsible in $log\ n$ hops because each hop cuts the distance to the destination by half. In the ring geometry, p can select its i-th neighbor from the peers whose distance from p clockwise in the circle is in the range $[(2i - 1)\ mod\ n, 2i\ mod\ n)$. This implies that in the ring geometry, each peer has $2i - 1$ options in selecting its ith neighbor. Thus in terms of flexibility of neighbor selection, there are a total of approximately $2^{(log\ n-1)*(log\ n-1)/2}$ options for each peer. The flexibility in route selection is approximately $(log\ n)!$ possible routes for a typical path.

Butterfly. The Butterfly geometry is an extension of the traditional butterfly network that supports the scalability requirements of DHTs. Viceroy [MNR02] is a DHT that uses this geo-

metry for efficient data location. The peers of a butterfly with size n are portioned into $log\ n$ levels and $n/log\ n$ rows. The peers of each row are subsequently connected to each other using successor/predecessor links. The number of peers in each row is $log\ n$, thus a sequential lookup in each row is done in $O(log\ n)$. In addition to successor/predecessor links, each peer has some links to the peers of other rows. The inter-row links are arranged in such a way that the distance between a peer in Level 1 of any row to any other row is $log\ n$. Routing a query in the Butterfly is done in three steps as follows. First, the query is sequentially forwarded to the peer that is at Level l of the same row as the query originator. This is done in $O(log\ n)$ routing hops. Second, from Level 1, the query is routed in $O(log\ n)$ routing hops to the row to which the destination peer belongs. Third, at the destination row, the query is traversed sequentially to the destination peers. Each of these steps is done in $O(log\ n)$ routing hops, thus the total time of query routing is $O(log\ n)$. The advantages of the Butterfly geometry is that the size of the routing table per peer, i.e. the number of neighbors of each peer, is a small constant number whereas, in most of other geometries, this size is $O(log\ n)$. However, the Butterfly geometry has poor neighbor and route selection flexibility, i.e. there is only one choice for selecting the neighbors or the route.

XOR. The XOR approach uses a symmetric unidirectional tree topology for structuring the peers of the P2P network. Kademlia [MM02] is a DHT that uses the XOR geometry for query routing. In Kademlia, the distance between two peers is computed as the numeric value of the exclusive OR (XOR) of their identifiers. Each peer p has $log\ n$ neighbors, where the i-th neighbor is any peer whose XOR distance from p is a value in $[2i, 2i + 1)$. Query routing proceeds by greedily reducing via intermediate peers the XOR distance from the destination peer. Kademlia provides the same neighbor selection flexibility as the tree geometry. In addition, in terms of route selection, the XOR geometry can reduce the bit differences in any order, and does not require strict left-to-right bit fixing as the tree geometry. Thus, it has a great route selection flexibility which is comparable to that of the ring geometry.

Hybrid. Hybrid geometries use a combination of geometries. Pastry [RD01] combines the tree and ring geometries in order to achieve more efficiency and flexibility. Peer identifiers are maintained as both the leaves of a binary tree and as points on a one-dimensional circle. In Pastry, the distance between a given pair of nodes is computed in two different ways : the tree distance between them and the ring distance between them. Peers have great flexibility of neighbor selection. For selecting their neighbors, peers take into account the proximity properties, i.e. they select the neighbors that are close to them in the underlying physical network. The route selection is also very flexible because, to route a message, peers have the possibility to choose one of the hops that do make progress on the tree or on the ring.

2.2.1.3 Super-peer

Unstructured and structured P2P networks are considered "pure" because all their peers provide the same functionality. In contrast, super-peer networks are hybrid between client-server systems and pure P2P networks. Like client-server systems, some peers, the super-peers, act as dedicated servers for some other peers and can perform complex functions such as indexing, query processing, access control, and metadata management. Using only one super-peer reduces to client-server with all the problems associated with a single server. Like pure P2P networks, super-peers can be organized in a P2P fashion and communicate with one another in sophisticated ways, thereby allowing the partitioning or replication of global information across all super-peers. Super-peers can be dynamically elected (e.g. based on bandwidth and processing power)

and replaced in the presence of failures. In a super-peer network, a requesting peer simply sends the request, which can be expressed in a high-level language, to its responsible super-peer. The super-peer can then find the relevant peers either directly through its index or indirectly using its neighbor super-peers. The main advantages of super-peer networks are efficiency and quality of service (i.e. the user-perceived efficiency, e.g. completeness of query results, query response time, etc.). The time needed to find data by directly accessing indices in a super-peer is very small compared with flooding. In addition, super-peer networks exploit and take advantage of peers' different capabilities in terms of CPU power, bandwidth, or storage capacity as super-peers take on a large portion of the entire network load. In contrast, in pure P2P networks, all nodes are equally loaded regardless of their capabilities. Access control can also be better enforced since directory and security information can be maintained at the super-peers. However, autonomy is restricted since peers cannot log in freely to any super-peer. Fault-tolerance is typically low since super-peers are single points of failure for their sub-peers (dynamic replacement of super-peers can alleviate this problem). Examples of super-peer networks include Napster], Publius [WRC00], Edutella [NWQ+02b][NSS03], and JXTA. A more recent version of Gnutella also relies on super-peers [ATS04].

Super-peer networks typically rely on some powerful and highly available peers, called super-peers, to index the data shared by peers which are connected to the system. Edutella is one of the most known super-peer networks. In Edutella, super-peers are arranged in the HyperCup topology [SSDN02], so messages can be communicated between any two super-peers in $O(logm)$ routing hops, where m is the number of super-peers. The process of joining a super-peer to the network consists of two parts : taking the appropriate position in the HyperCuP topology and announcing itself to its neighbors. Each ordinary peer joins the system by connecting to a super-peer. To support efficient query routing, at each super-peer, two kinds of routing indices are maintained : super-peer/peer (SP/P) indices and super-peer/super-peer (SP/SP) indices. Queries are routed over super-peers by using the SP/SP indices, and to ordinary peers based on the SP/P indices. In the SP/P indices, each super-peer stores information about the characteristics of the data which are shared by the peers that are connected to it. These indices are used to route a query from the super-peer to its connected peers. At join time, peers provide their metadata information to their super-peer by publishing an advertisement. To index the provided metadata, Edutella uses the schema-based approaches which have successfully been used in the context of mediator-based information systems (e.g. [Wie92]). To ensure that the indices are always up-to-date, peers notify super-peers when their data change. When a peer leaves the network, all references to this peer are removed from the indices. If a super-peer fails, its formerly connected peers must connect to another super-peer chosen at random, and provide their metadata to it. The second type of indices is SP/SP indices which are essentially summaries (possibly also approximations) of SP/P indices. Update of SP/SP indices is triggered after any modification to SP/P indices as follows. When a super-peer changes its SP/P index, e.g. due to a peer's join/leave, it broadcasts an announcement of update to the super-peer network by using the HyperCuP protocol. The other super-peers update their SP/SP indices accordingly. Although such a broadcast is not optimal, it is not too costly either because the number of super-peers is much less than the number of all peers. Furthermore, if peers join/leave frequently, the super-peer can send a summary announcement periodically instead of sending a separate announcement for each join/leave. The query routing in Edutella is done as follows. When a peer receives a query issued by the user, it sends the query to its super-peer. At the super-peer, the metadata used in the query are matched against the SP/P indices in order to determine

Table 2.3 – Comparison of P2P networks

Requirements	Unstructured	Structured	Super-peer
Autonomy	high	low	moderate
Query expressiveness	"high"	"low"	high
Efficiency	low	high	high
QoS	low	high	high
Fault-tolerance	high	high	low
Security	low	low	high

local peers which are able to answer the query. If the query cannot be satisfied by local peers, it is forwarded to other super-peers using SP/SP indices.

2.2.2 Comparing P2P Networks

From the perspective of data management, the main requirements of a P2P network are [DGY03] : autonomy, query expressiveness, efficiency, quality of service, fault-tolerance, and security. We describe these requirements in the following. Then, we compare P2P networks based on these requirements.

- Autonomy : an autonomous peer should be able to join or leave the system at any time without restriction. It should also be able to control the data it stores and which other peers can store its data, e.g. some other trusted peers.
- Query expressiveness : the query language should allow the user to describe the desired data at the appropriate level of detail. The simplest form of query is key look-up which is only appropriate for finding files. Keyword search with ranking of results is appropriate for searching documents. But for more structured data, an SQL-like query language is necessary.
- Efficiency : the efficient use of the P2P network resources (bandwidth, computing power, storage) should result in lower cost and thus higher throughput of queries, i.e. a higher number of queries can be processed by the P2P system in a given time.
- Quality of service : refers to the user-perceived efficiency of the P2P network, e.g. completeness of query results, data consistency, data availability, query response time, etc.
- Fault-tolerance : efficiency and quality of services should be provided despite the occurrence of peers failures.
- Security : the open nature of a P2P network makes security a major challenge since one cannot rely on trusted servers. Wrt. data management, the main security issue is access control which includes enforcing intellectual property rights on data contents.

Table 2.3 summarizes how the requirements for data management are possibly attained by the three main classes of P2P networks. This is a rough comparison to understand the respective merits of each class. For instance, "high" means it can be high. Obviously, there is room for improvement in each class of P2P networks. For instance, fault-tolerance can be made higher in super-peer by relying on replication and fail-over techniques.

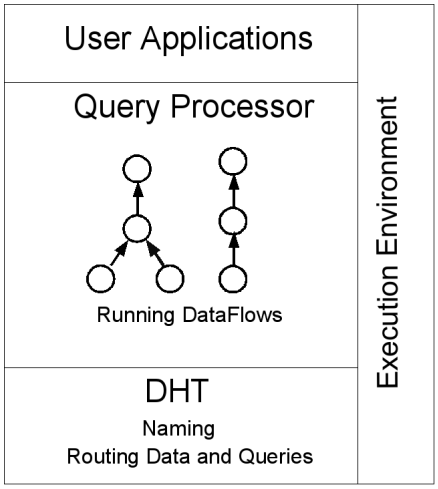


Figure 2.10 – Pier Architecture

2.2.3 Complex Query Processing

Although highly efficient, P2P systems based on DHTs provides only exact-match data lookups. This compromise their use in applications where more advanced query facilities are required. Examples of complex queries are : join queries and range queries. In this section, we discuss the techniques proposed to support these complex queries in DHT-based P2P systems.

2.2.3.1 Join Queries

PIER. The first attempt to execute complex SQL queries on top of a DHT is PIER [HHL+03]. PIER organizes the nodes using Bamboo, a DHT loosely based on Pastry [RD01], where the nodes cooperates to evaluate a join query, although PIER is DHT agnostic since other DHTs including CAN [RFH+01] and Chord [SMK+01] have been used in different stages of its development. The join algorithms are adaptations of textbook parallel and distributed schemes that have been adapted to suit the DHT scenario.

PIER is a three-tier system composed by a execution environment, a DHT and a query processor as shown in Figure 2.10.

The execution environment has been designed based on the key idea that both simulation and physical environments use the same code allowing to test network and multiple machines in simulations running the real system code. The PIER programming model is event-based, the events are processed by a single thread which is justified because it allows to utilize asynchronous I/O and because it fits naturally with discrete-event network simulation. The execution environment is key in the design of PIER making more easier the early identification of bugs in the distributed logic of query execution.

The DHT is used both for indexing and storage. Traditional *get* and *put* methods are provided as well as others methods in order to perform maintenance operations. In PIER, tuples and queries have a :

- *namespace*, used in query processing to identify the table a tuple belongs to. In the case of a query, it is used to identify a multicast group where the query is sent.

- *resourceID*, composed of one or more attributes of a tuple, it can be viewed as a partitioning key. In the case of a query, it is used to assign a globally unique identifier.
- *instanceID*, is an identifier chosen at random by the user application in order to separate items that have been stored using the same *namespace* and *resourceID*.

The DHT key is calculated via a hash function on *namespace* and *resourceID*. Thus, tuples having the same *namespace* and *resourceID* are stored at the same node. Once a tuple arrives to a node a *newData* callback is used to notify this event. PIER does not provide persistent storage, instead it uses the concept of *soft state* where a node stores tuples in main memory for a time period after which the tuples is discarded. If a node wishes extend the period a tuple is stored, it must invoke the *renew* function. PIER does not maintain system metadata. As a result, every tuple in PIER is self-describing, containing its table name, column names, and column types.

The PIER query processor is a "boxes-and-arrows" dataflow engine. Using a graphical interface called LightHouse a query plan is constructed by arranging boxes representing query operators joined by directed arcs representing the data flow between operators. A query is sent by an user application to a PIER node where the query is parsed , disseminated and the query results are forwarded to the user application. This PIER node is called the proxy.

In the query parsing the proxy node generates a representation of the query in terms of Java objects suitable for the query executor. Since there is no system catalog the parser does not check the existence or type of columns referenced in the query. The proxy disseminates the query contacting nodes, using a *multicast* communication primitive, that hold data needed to process the query. Instances of each operator and dataflow links are created at each node receiving the query. Access to tuples stored locally is provided by the *lscan* iterator. When a result is generated, it is delivered to the proxy node. All the functions implemented by the DHT, as listed in Table 2.4, provide an useful API to applications.

Table 2.4 – API provided by the DHT

item *get*(namespace, resourceID)
put(namespace, resourceID, instanceID, item, lifetime)
boolean *renew*(namespace, resourceID, instanceID, item, lifetime)
iterator *lscan*(namespace)
multicast(namespace, resourceID, item)
item *newData*(namespace)

To determine the subset of network nodes needed to process a query PIER uses three kinds of indexes : a broadcast-predicate index, an equality-predicate index, and a range-predicate index. A broadcast-predicate index is based on a distribution tree and is used with queries that range over all the data to find all the data. The equality-predicate index is used with queries that need to find a specific value, this is supported directly by using the DHT primitives. A range-predicate index is based on a Prefix Hash Tree [RRHS04] a distributed data structure that supports range queries over DHTs.

The execution of binary join algorithms is the core functionality in PIER. The join algorithms implemented in PIER are a version of the *symmetric hash join* and a variant of a distributed join algorithm called *Fetch Matches*. In the *symmetric hash join* tuples are read when appear at either input, added to the corresponding hash table and probed against the opposite hash table based on the tuples received so far. In PIER the tuples belonging to different relations, for instance R and S, are supposed to be horizontally partitioned and stored in the DHT under

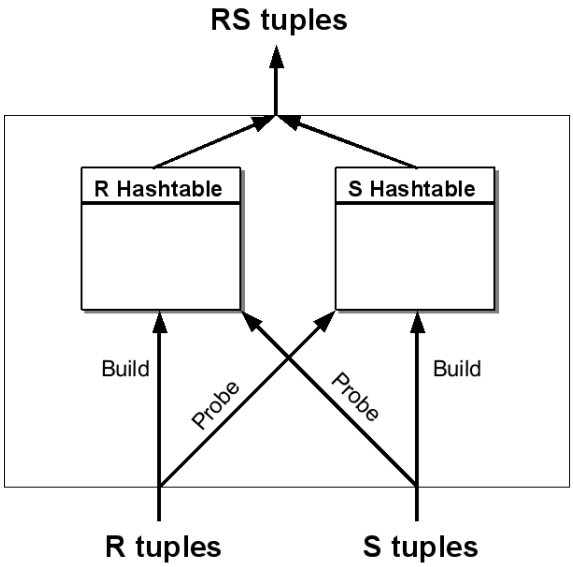


Figure 2.11 – Symmetric or Doubly pipelined hash join

separate namespaces N_R and N_S. A query Q is disseminated to all nodes that store tuples in the namespaces involved in the query. Each node receiving the query scan its relations using *lscan* searching for tuples that satisfy the selection predicates. All these tuples are rehashed, using the value of the join attribute to form the *resourceID*, in a new namespace N_Q where they are processed. When a tuple is rehashed, the *put*(namespace, resourceID, instanceID, item, lifetime) function is used where namespace is N_Q, resourceID is the concatenation of join attribute values, instanceID is chosen at random, item is composed by the selection attributes and additional information specifying the namespace the tuple comes from, and time is the amount the time the item will be available in the N_Q namespace. Thus, all the tuples with the same *namespace* and *resourceID* are stored on the same node and process using the *doubly pipelined hash join* as shown in Figure 2.11. The Algorithm 1 shows the steps followed by a node in N_R. The same steps are executed in a node containing tuples of relation S. When a tuple from R(or S) arrives to a node in the namespace N_Q a *newData* callback is issued and the *get* function is invoked in order to find matches in the other relation. Whenever matching tuples are found, they are sent to the next stage in the query or they are immediately sent to the query initiator using the function *send*. The Algorithm 2 shows the steps followed by a node in N_Q.

Algorithm 1 Build Hash Table : A node that stores R tuples scans and rehashes tuples using the join attribute value

Require: **Q**, a join query.

1: **for** each $r \in R$ **do**
2: $resourceID = getJoinAttributeValue(r, Q)$
3: $instanceID = random()$
4: $item = (getSelectAttributes(r, Q), R)$
5: $put(N_Q, resourceID, instanceID, item, time)$
6: **end for**

Algorithm 2 Probe : A node in N_Q that stores R tuples receives a *newData* callback and searches locally for S tuples

Require: N_Q, the namespace of the query. **r**, an R tuple.

1: $s = get(N_Q, resourceID)$
2: **if** probe(r,s) **then**
3: $result = createResult(getSelectAttributes(r, Q), getSelectAttributes(s, Q))$
4: $send(Query_{initiator}, resourceID, instanceID, result, time)$
5: **end if**

Let us consider the join query q over relations R and S with schema (a, b, c) :

q : Select R.b,S.c
From R,S
Where R.a=S.a

We assume that the tuples belonging to R and S have been partitioned into two namespaces N_R and N_S as is shown in Figure 2.12. The query initiator disseminates the query to nodes in namespaces and all the nodes that store tuples run scan/rehash steps as follows. All the tuples stored locally are retrieved using *lscan* and rehashed based on the join attribute value and the name of a new namespace N_Q using the *put* function (see Figure 2.12). Considering the query q, tuples with the same value on $R.a$ and $S.a$ are rehashed to the same node in N_Q. Nodes in N_Q use *newData* and *get* in order to obtain join results as follows. Once a rehashed tuple arrives on one of the relations, the query processor is informed by a *newData* callback that triggers a call to the *get* function (expected to be local) in order to find matches in the other relation. Note that tuples from the relations arrive in an arbitrary interleaved fashion thus a symmetric join operator (see Figure 2.11) is implemented locally in each N_Q node. Once a match is found, tuples are concatenated and sent to the query initiator.

The rehashing step over both tables may need much bandwidth during the execution of the *symmetric hash join*. To reduce this, the followings improvements of the *symmetric hash join* are proposed : *Fetch Matches*, *symmetric semi join* and *Bloom joins*. In order to avoid the rehashing of all the tuples, the *Fetch Matches* algorithm, a variant of a traditional distributed join algorithm [ML86], is implemented. This algorithm assumes that one of the relations, say S, is already hashed on the join attributes. All the nodes in N_R are scanned using *lscan* and, for each tuple of R, the DHT is queried using the *get* function, for S tuples matching $R.a$. Since the *get* invocation are done at the DHT level, selection and projection over the selected S tuples cannot be performed at the remote site. Thus, selection and projection are applied at N_R nodes and join results are sent to the query initiator. The *symmetric semi join* is a DHT-base version of a traditional query rewrite strategy. This algorithm reduces the bandwidth, projecting tuples locally, at N_R and N_S nodes, to only hash key and join attributes. A *symmetric hash join* is run on the projected tuples and the resulting tuples are used as source for two *Fetch Matches* joins to retrieve the other attributes specified in the query ($R.b$ and $S.c$ in the case of the query q of Figure 2.12). *Bloom joins* is a rewrite strategy that reduces the amount of tuples rehashed in the *symmetric hash join*. Each node creates a local Bloom filter, of its local R or S fragment, which is publied into a temporary DHT namespace for each table. All the nodes in the namespaces cooperate to generate Bloom filters $B_{R.a}$ and $B_{S.a}$ on all of R and S, respectively, which are broadcast to all the nodes storing the opposite table. Once a node receives a Bloom filter, it

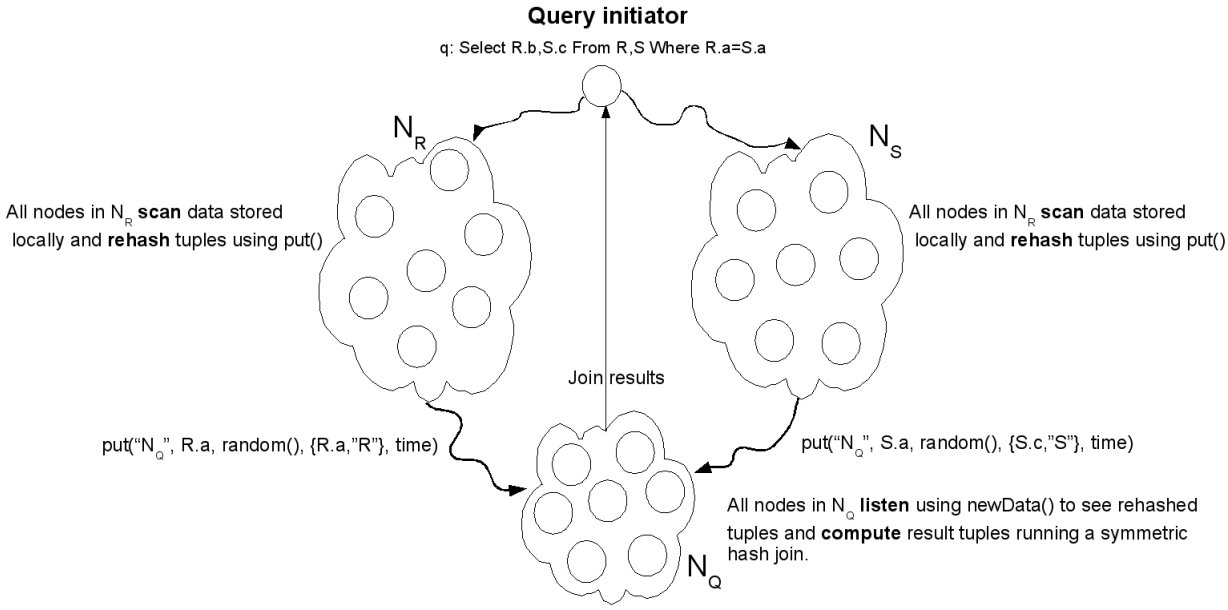


Figure 2.12 – Execution of a join query using PIER

retrieves its tuples stored locally using *lscan* and rehashes only those tuples that match the filter. The possible false positives are controlled by the *symmetric hash join* which completes the join.

RJoin. This algorithm is designed for the evaluation of continuous multi-way join queries on top of DTHs. RJoin [ILK08] is based on the idea of incremental evaluation : as relevant tuples arrive continuously, a multi-join query is rewritten continuously by replacing the value of the join attributes of arriving tuples in the query, a new query with fewer joins is created and is assigned to different nodes of the network. Thus, RJoin distributes the responsability of evaluating the query to many network nodes. RJoin organizes the nodes in a Chord ring where the nodes insert (index) tuples, pose continuous queries and participe in query processing tasks (see Figure 2.13).

In Rjoin, tuples follow the relational data model. A timestamp, $pubT(s)$, is assigned to each tuple s with the time that the tuple was inserted into the network by some node. Queries are expressed using SQL and timestamped with the time that the query was submitted to the network. The timestamp of a query is denoted by $insT(q)$.

The indexing of tuples is based on a variation of hash partitioning. RJoin indexes tuples for streams $S = \{S_1, S_2, ..., S_m\}$ as follows. Let s be a tuple belonging to S_i. Let $A = (A_1^i, A_2^i, ..., A_k^i)$ be the set of attributes in s and $val(s, A_j^i)$ be a function that returns the value of the attribute $A_j^i \in A$ in tuple s. A node indexes a tuple using each attribute name and each attribute value it has. For each attribute A_j^i a node generates two identifiers : $AIndex_j = Hash(S_i + A_j^i)$ to index a tuple at the attribute level and $VIndex_j = Hash(S_i + A_j^i + val(s, A_j^i))$ to index a tuple at the value level. Function $Hash()$ is used to generate an identifier that allows to index the tuple s to the first node which is equal or follows the identifier clockwise in the identifier space. For each $AIndex_j$ the node creates a message $newTuple = (s, AIndex_j, attribute, pubT(s))$. In a similar way, for each $VIndex_j$ a message $newTuple = (s, VIndex_j, value, pubT(s))$ is created.

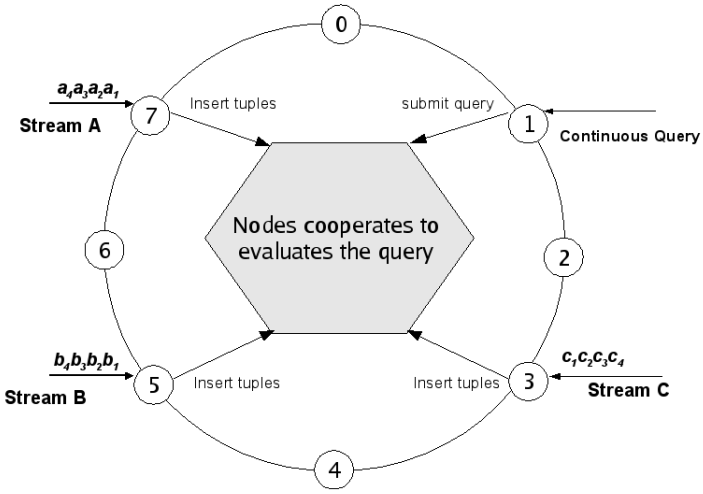


Figure 2.13 – Distributed evaluation of multi-way join queries in RJoin

Therefore, in a Chord ring of N nodes, RJoin creates $2k$ messages to index a tuple in $2kO(logN)$ hops. The Algorithm 3 shows how a tuple is indexed.

Algorithm 3 Insert_Tuple(s)

Require: **s**, a tuple $\in S_i$ of k attributes $A = (A_1^i, A_2^i, ..., A_k^i)$ to be indexed into the network.

1: $M = \varnothing, I = \varnothing$
2: **for** $j = 0$ to k **do**
3: $AIndex_j = Hash(S_i + A_j^i)$
4: $newTuple = (s, AIndex_j, attribute, pubT(s))$
5: $I = I \cup \{AIndex_j\}$
6: $M = M \cup \{newTuple\}$
7: $VIndex_j = Hash(S_i + A_j^i + val(s, A_j^i))$
8: $newTuple = (s, VIndex_j, value, pubT(s))$
9: $I = I \cup \{VIndex_j\}$
10: $M = M \cup \{newTuple\}$
11: **end for**
12: multiSend(M,I)

To send messages created by the indexing of tuples, RJoin uses an extension of the standard API of the Chord protocol. This extension allows a node to send a set of M messages to a set of I identifiers using the function $multiSend(M, I)$. This function is similar to the Chord function $lookup()$. Basically, the function $multiSend(M, I)$ sends each message M_j to the node responsible for I_j in the Chord ring, where $1 \leq j \leq k$. The function $multiSend(M, I)$ works as follows. The first node that invokes $multiSend(M, I)$ sorts the set I in ascending order clockwise starting from its own identifier. This node searches for the node responsible of $head(I)$ (the first element of I) and sends a $multiSend()$ message. When a node, with identifier id in the Chord ring, receives a $multiSend()$ message, it compares its own identifier with $head(I)$. If $id < head(I)$ the node just forwards the $multiSend()$ message. If $id \geq head(I)$, then the node process the message M_j corresponding to $head(I)$. This node creates a new set $I^{'}$ from I,

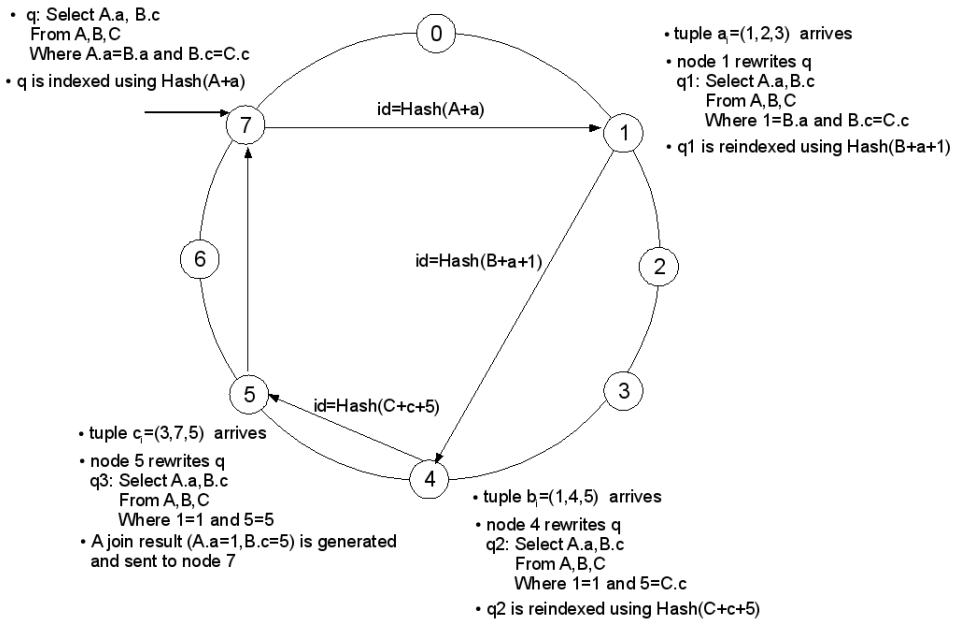


Figure 2.14 – Evaluation of a query in RJoin

deleting all the elements I_j such that $head(I) \leq I_j \leq id$, since it is responsible for them. All the M_j messages corresponding to the deleted I_j elements are processed and M is updated creating $M^{'}$. Subsequently, the node invokes $multiSend(M^{'}, I^{'})$ and this procedure continues until the set I is empty.

Regarding queries, once a query is posed in the network it is indexed to a node where it is stored, then waiting for tuples. A query is indexed at the attribute and value level, however once a query is submitted it is indexed at the attribute level. Thus, a query meets tuples indexed with the same attribute, is rewritten with the attribute values present in the arriving tuples and reindexed to another node where it waits for tuples and possibly will be reindexed until no more join exists in the Where clause of the query. Note that, when the join query is rewritten and reindexed, it is reduced to a simple select-project query. For example, consider the following query and three streams A,B and C with the same schema (a, b, c). :

q : Select A.a,B.c
From A,B,C
Where A.a=B.a and B.c=C.c

We assume that the query is submitted by node 7 and indexed at the attribute level. Node 7 uses the function $nextKey(q)$ to obtain the relation-attribute pair or relation-attribute-triple used to index query q. Thus, q is indexed using $Hash(A + a)$ that generates an identifier that allows to index the query q to the first node which is equal or follows the identifier clockwise in the identifier space as is shown in Figure 2.14. At node 1, the query is stored and waits for arriving tuples. Tuple $a_i = (1, 2, 3) \in A$ indexed at the attribute level using $Hash(A + a)$ arrives to node 1. The arriving tuple is used to rewrite the query using the value of attribute a only if $pubT(a_i) \geq insT(q)$. If so, the query is rewritten by node 1, replacing the expression $A.a$ of the query predicate by the value of attribute a of the arriving tuple, generating $q1$:

Select A.a,B.b From A,B,C Where 1=B.a and B.c=C.c. Node 1 uses $nextKey(q1)$ and $q1$ is reindexed using predicate $1 = B.a$. To send the query to the next node, node 1 creates a message $Eval = (q1, key, owner, insT(q1))$, where $key = B + a + 1$ is the relation-attribute-value triple used as the key used to determine the node that receives the rewritten query and *owner* is the identifier of the node that submitted the query (node 7). Thus, as shown in Figure 2.14, query $q1$ is reindexed to node 4 where tuples belonging to stream B, that have the value 1 in its attribute a, are indexed at the value level using $B + a + 1$. Node 4 stores query $q1$ and each arriving tuple is used to rewrite it and store it locally since they may be used in the future when an another rewritten query arrives. Assuming that at node 4 a tuple $b_i = (1, 4, 5) \in B$ arrives, it is used to rewrite query $q1$, generating $q2$: Select A.a,B.b From A,B,C Where 1=1 and 5=C.c and simplifying the query predicate. Node 4 reindexes $q2$ to node 5 (see Figure 2.14) using $Hash(C + c + 5)$ and creates the message $Eval = (q2, key, owner, insT(q2))$, with $key = C + c + 5$. At node 5 query $q2$ is rewritten using tuples belonging to stream C, with value 5 in its attribute c. At this point, the query predicate is totally simplified and a join result is generated and sent to node 7.

Algorithm 4 Process_indexedTuple(newTuple)

Require: newTuple, a newTuple message containing $(s, AIndex_j, LEVEL, pubT(s))$

```
1:  M = ∅, I = ∅
2:  for i = 0 to sizeof(queryTable) do
3:    if (pubT(s) ≥ insT(q_i) then
4:      q' = rewrite(q_i, s)
5:      if (JoinPredicate(q') is totally simplified) then
6:        result = createResult(q_i)
7:        sendResult(result,owner(q_i))
8:        continue
9:      end if
10:     key = nextKey(q')
11:     id = Hash(key)
12:     Eval = (q', key, owner)
13:     I = I ∪ {id}
14:     M = M ∪ {Eval}
15:   end if
16: end for
17: if LEVEL==Value then
18:   store s locally
19: end if
20: multiSend(M,I)
```

In general, tuples are indexed at the attribute level in order to trigger the evaluation of a query and are indexed at the value level in order to rewrite queries and simplify the join predicate. Tuples indexed at the value level are stored locally since future queries will be rewritten using it. The Algorithm 4 shows the steps followed by a node when it receives a tuple indexed at the attribute or at the value level. When a node receives a rewritten query, it is stored locally where it waits for arriving tuples. However, a node can store tuples that have a timestamp $pubT(s) \geq insT(q)$ which have been already stored. Using each of such tuples, the query is

rewritten and reindexed to another node. The Algorithm 5 shows the steps followed by a node when receives a rewritten query.

Algorithm 5 Process_Query(q)

Require: Eval, a Eval message containing $(q, key, owner, insT(q))$

```
1: M = ∅, I = ∅
2: T list of tuples s that match q
3: store q in the queryTable
4: for i = 0 to sizeof(T) do
5:    if (pubT(s_i) ≥ insT(q_i) then
6:       q' = rewrite(q_i, s)
7:       if (JoinPredicate(q') is totally simplified) then
8:          sendResult(result,owner(q_i))
9:          continue
10:      end if
11:      key = nextKey(q')
12:      id = Hash(key)
13:      Eval = (q', key, owner)
14:      I = I ∪ {id}
15:      M = M ∪ {Eval}
16:   end if
17: end for
18: multiSend(M,I)
```

Since data streams are potentially unbounded in size, it is not possible to store locally all the arriving tuples indexed at the value level. To deal with infinite inputs, RJoin supports time-based and tuple-based sliding windows.

In the previous example (see Figure 2.14) the joins of the query are evaluated in the order they appear in the Where clause. However, similar to traditional distributed techniques to execute joins, intermediate results can lead to increase the query processing load and network traffic. In Rjoin, each *newTuple* message produces the rewriting and reindexing of a query. Thus, a query indexed using a relation-attribute pair with tuples arriving at a high rate leads to an increase of query rewriting and reindexing. Similarly, when a node receives a query rewritten using a relation-attribute-value triple with tuples containing the same attribute value arriving at a high rate the query rewriting an reindexing increases. This is a problem of join ordering that RJoin solves indexing a query using a relation-attribute or a relation-attribute-value with an arrival rate predicted to be low. To this end, before indexing a query, RJoin collects information about the rate of incoming (RIC) tuples of the nodes to which the query could be indexed and then the node with the smaller RIC is chosen to index the query. For example, considering the query of Figure 2.14 being indexed by a node n, the candidate nodes where the query could be indexed are $n_1 = Hash(A + a)$, $n_2 = Hash(B + a)$, $n_3 = Hash(B + c)$ and $n_4 = Hash(C + c)$. RJoin uses the $multiSend(msg, I)$ function to collect RIC information, where msg is used to store the arrival's rate and IP addresses and $I = \{n_1, n_2, n_3, n_4\}$. If we consider that the ascending order of identifiers is (n_1, n_2, n_3, n_4) the RIC information is collected as follows. The node n_1 is the first node contacted, n_1 store the arrival rate of tuples belonging to stream A and its address IP in msg. Subsequently, n_1 forwards msg to n_2 that stores its information about the arrival rate

of tuples belonging to stream B and its address IP. Similarly, n_2 forwards the message to n_3. Finally, n_4 receives the message from n_3 and sends all the RIC information and IP addresses collected to node n. Thus, node n indexes the query to the node with the smaller arrival rate in one hop. The total cost of this operation is $O(klogN) + 2$ messages, where k is the number of nodes where the query could be indexed. RJoin assumes that the collected information is similar for the future or at least during a reasonable window time. Thus, the cost of collect RIC information is payed only once for each indexed query and the benefits of keeping the cost of rewriting queries at low levels are perceived with every tuple insertion.

2.2.3.2 Range Queries

Conventional structured peer to peer systems based on distributed hash tables (CAN [RFH+01], Chord [SMK+01], Pastry [RD01], etc.) are not effective for supporting data partitioning and retrieval based on ranges since hashing destroys the ordering of data. A range query is a query asking for all objects with values in a certain range. Range queries are present naturally in many applications domains such as internet databases [HHL+03], moving objects databases [HRM08] and scientific computing [WS92].

Different approaches to the problem of processing range queries in P2P systems have been proposed. These approaches can be divided into those that rely on an underlying DHT and those that do not. In the first group, a P2P system based on Locality Sentitive Hashing is proposed in [GAA03] where similar ranges are hashed to the same DHT node with high probability. However, this method can only obtain approximate answers. The P-tree [CLGS04] structure uses Chord as its overlay routing architecture. The key idea of a P-tree is to maintain parts of semi-independents B+-trees at each peer. Data is stored in leaf nodes that form a Chord ring. Exact and range queries take $O(logn)$ steps but its performance degrades when the data is skewed. Prefix Hash Tree (PHT) [RRHS04] hashes the prefix labels of PHT nodes over the DHT identifier space. In the second group, P2P systems based on Skip Graphs [HJS+03] can support range queries but they do not guarantee data locality and load balancing in the whole system. BATON (BAlanced Tree Overlay Network) [JOV05] is based on a binary balanced tree structure providing scalability and robustness similar to that of the B-tree. In BATON each node of the tree is maintained by a peer. Range queries are answered in $O(logn + X)$ steps. In the following we present in more detail how range queries are answered in the PHT and BATON.

Prefix Hash Tree. The Prefix Hash Tree (PHT) is a trie-based distributed data structure that supports range queries over a DHT. PHT assumes that the data being indexed are binary strings of length D. PHT cannot by itself protect against data loss when nodes go down. However, the failure of a node does not affect the availability of any other node in the trie. PHT is agnostic to the choice of the underlying DHT since it is built entirely on top of the $lookup(key)$ operation.

A PHT is a binary trie over the data set where the left branch of a node is labeled 0 and the right branch is labeled 1. Thus, a node n in a PHT represents a prefix bit string, called label l, produced by the concatenation of the labels of all branches in the path from the root to n.

A PHT is based on the following properties :

1. **Universal prefix.** Each node has either 0 or 2 children.
2. **Key storage.** A key k is stored at a leaf node whose label is a prefix of k.
3. **Split.** Atmost B keys are stored at each leaf node.
4. **Merge.** A sub-tree of a internal node contains at least $B + 1$ keys.

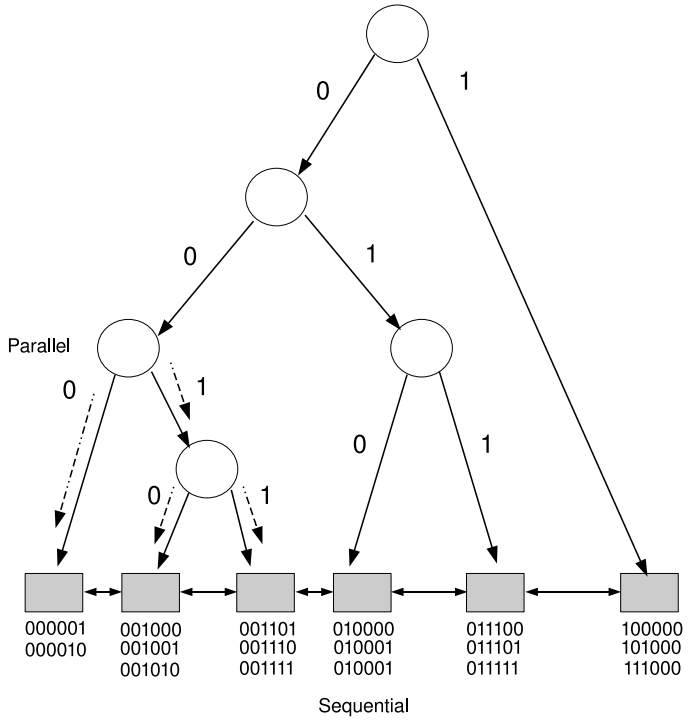


Figure 2.15 – Prefix Hash Tree

5. **Threaded leaves.** Each leaf node maintains a pointer to its left and right adjacent leaf node respectively.

These properties are invariants in a PHT. Property 1 assures that each node has exactly one prefix. Property 2 states that a key k is only stored at $leaf(k)$. Properties 3 and 4 dictate how the PHT adapts to the distribution of keys in the data set. Finally, property 5 ensures the construction of a double linked list formed by the leaves of the PHT. This linked list allows the sequential traversal of leaves for answering range queries. PHT vertices are assigned to DHT nodes by hashing the prefix label of a PHT node over the DHT identifier space. The peer responsible for the label l is the peer whose identifier is closest to $hash(l)$. Thus, PHT nodes are assigned to DHT nodes. This implies that given a label l, it is possible to locate its corresponding PHT node via a single DHT lookup.

Figure 2.15 is an example of a PHT with $D = 6$ and $B = 4$. Shadow nodes are leaf nodes storing the keys and forming a double linked list. As an example, the key 010000 is stored at $leaf(010000) = 010*$.

In a PHT, the lookup operation $lookup(k)$ returns the unique leaf node $leaf(k)$ whose label is a prefix of k. Given a key k of length D, there are $D + 1$ distinct prefixes of k. The obvious algorithm is to obtain $leaf(k)$ performing a linear scan of these potential nodes $D + 1$ as shown in algorithm 6. In the case of the PHT of Figure 2.15, given the key $k = 001100$, the prefixes 0, 00, 001 and 0011 are tested to obtain the leaf node that stores k.

However, considering that a PHT is a binary trie, the linear scan can be improved imple-

Algorithm 6 Linear lookup

Require: **k**, a key.

```
1: for i = 0 to D do
2:    node = lookup(Prefix_i(k))
3:    if node is a leaf node then
4:       return node
5:    end if
6: end for
```

menting a binary search on prefix length. This reduces the number of DHT lookups from $D+1$ to $logD$.

Recall that a PHT is built entirely on top of the lookup operation. Thus, once this operation is implemented we can show how a PHT supports range queries. Given two keys a and b such as $a \leq b$, a range query returns all keys k satisfying $a \leq k \leq b$. In a PHT two algorithms for range queries are implemented.

1. Sequential. Using the PHT lookup operation implemented as a binary search. This algorithm searches $leaf(a)$ and scans sequentially the linked list of leaf nodes until the node $leaf(b)$ is reached.
2. Parallel. This algorithm identifies the node whose label corresponds to the smallest prefix range that completely covers the range $[a, b]$. To reach this node a simple DHT lookup is used and the query is forwarded recursively to those children which overlap with the range $[a, b]$.

Consider the PHT of Figure 2.15 and a query for the range $[000001, 001111]$. The sequential algorithm uses the PHT lookup operation to locate $leaf(000001)$. Once the leaf node is located, the double linked list formed by the leaves nodes is traversed until the node $leaf(001111)$ is reached. In the parallel algorithm, the prefix $00*$ is identified as the smallest prefix range that completely covers the range $[000001, 001111]$. A DHT lookup is used to directly reach this node, after which the query is forwarded recursively to those children which overlap with the range specified in the query. Figure 2.15 shows how the query is forwarded in parallel. In both cases, the response to the query is composed by the set of keys $\{000001, 000010, 001000, 001001, 001010, 001101, 001110, 001111\}$.

BATON. The first attempt to build a P2P overlay network based on a balanced tree structure is BATON. In BATON each node of the tree is maintained by a peer. The position of a node is determined by a (level,number) tuple, of which level starts by 0 at the root and number starts from 1 at the root and sequentially assigned in an in-order traversal. Each tree node stores links to its parent, children, adjacent nodes and selected neighbor nodes which are nodes at the same level. Two routing tables, a *left routing table* and a *right routing table*, store links to the selected neighbor nodes. For a node numbered i, these routing tables contains links to nodes located at the same level with numbers that are less (left routing table) and greater (right routing table) than i by a power of 2. The j^{th} element in the left (right) routing table at node i contains a link to the node numbered $i - 2^{j-1}$ (respectively $i + 2^{j-1}$) at the same level in the tree. This structure is in the spirit of Chord except that the geometry is on a straight line rather than on a circle. Figure 2.16 shows the routing table of node 6.

In BATON, each node, both leaf and internal, is assigned a range of values. For each link this range is stored at the routing table and when it range changes, the link is modified to record the change. The range of values managed by a node is required to be to the right of the range

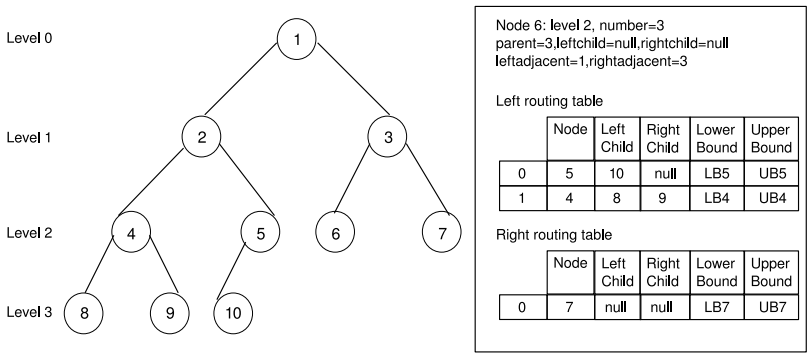


Figure 2.16 – BATON structure-tree index and routing table of node 6

managed by its left subtree and less than the range managed by its right subtree (see Figure 2.17). Thus, BATON builds an effective distributed index structure.

When a node joins or leaves the system can cause the tree to become imbalanced. The algorithm to join a node seeks out leaf nodes with its routing tables full and having less than two children. If the routing tables of a node are full, the level is full and the node joining the system can be accepted as a child, otherwise it forwards the request to its parent. In a network of n nodes, the height of the tree is $O(logn)$, thus a join request cannot be forwarded more than $O(logn)$ times. When a node accepts a new node as its child, it contacts all neighbor nodes in its routing tables to inform about the new node and updates its adjacent links accordingly. Thus, a node joining the system will not upset the tree balance. A leaf node can leave the system safely when there is no neighbor node in its routing tables with children. Otherwise, it contacts a child node of one of its neighbor nodes to replace its position. If the leaving node is not a leaf node, it must find a node to replace it by contacting to one of its adjacent nodes. Considering that the process of finding a replacement node always goes down, it is bounded by the height of the tree which is $O(logn)$. When the join or leave is part of a load balancing process the above approach may not be permitted. In this case, BATON restructures the tree using rotations in a similar way to AVL trees [JOV05].

A range query can be processed as follows. For a range query Q with range $[a, b]$ submitted by node i, it looks for a node that intersects with the lower bound of the searched range. The peer that stores the lower bound of the range checks locally for tuples belonging to the range and forwards the query to its right adjacent node. In general, each node receiving the query checks for local tuples and contacts its right adjacent node until the node containing the upper bound of the range is reached. Partial answers obtained when an intersection is found are sent to the node that submits the query. In a network of n nodes, the first intersection is found in $O(logn)$ steps using an algorithm for exact match queries [JOV05]. Therefore, a range query with X nodes covering the range is answered in $O(logn + X)$ steps. The Algorithm 7 shows the steps followed to process a range query.

As an example, consider the query Q with range $[7, 45]$ issued at node 7, it executes an exact match query looking for a node containing the lower bound of the range (see dashed line in Figure 2.17). Since the lower bound is in the range assigned to node 4, it checks locally for

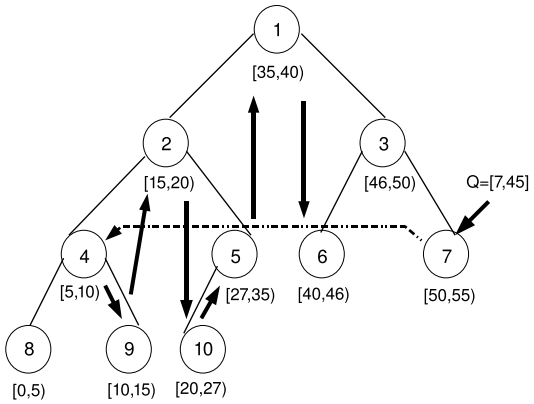


Figure 2.17 – Range query

Algorithm 7 Range Query

INPUT : Q, a range query in the form [a,b].
OUTPUT : T, tuples belonging to range [a,b].

{% Searching for the node storing the lower bound of the range}
1: **At query originator node do**
2: {
3: p = ExactQuery(a)
4: send query to node p
5: }
{% A node p receiving Q searches for local tuples and sends Q to its right adjacent node}
6: **At each node p that receives Q**
7: {
8: T = Range(p) ∩ [a,b]
9: send T to query initiator
10: **if** Range(RightAdjacent(p))∩ [a,b] ≠ ∅ **then**
11: send Q to right adjacent node of p
12: **end if**
13: }

tuples belonging to the range and forwards the query to its adjacent right node (node 9). Node 9 checks for local tuples belonging to the range and forwards the query to node 2 as is shown in Figure 2.17. Nodes 10, 5, 1 and 6 receive the query, they check for local tuples and contact its respective right adjacent node until the node containing the upper bound of the range is reached.

2.2.4 Conclusion

Unstructured P2P networks typically use a simple flooding scheme which is inefficient in terms of response time and consumes much network traffic. Furthermore, they are not suitable for efficient processing of continuous queries as they do not provide guarantees of any kind. Structured networks (i.e. DHT) provide more efficient key-based search. Because applications that process streams from different sources are inherently distributed and because distribution is a well accepted approach to improve both performance and scalability [CG07][TcZ07] of a DSMS, using a DHT is a natural choice to face the challenges motivated by the processing of continuous join queries.

In the context of join queries, the most relevant state of the art approaches, PIER [HHL$^+$03] and RJoin [ILK08], rely on a hash function to distribute data streams in the DHT using join attribute values. However, it is well known that hash-based join algorithms suffer from join attribute skew, i.e. certain join attribute values are much more frequent than others [ÖV99], which hurts load balancing and thus response time. It is important to note that data skew occurs naturally in many data streaming applications [XKZC08]. For example, in online analysis of transaction logs generated by telephone call records, some numbers used for online tv contests register a huge number of phone calls. In network monitoring applications, malicious traffic traces show that an abnormally high number of source addresses are connected to a single destination address.

Regarding the indexing of tuples, PIER [HHL$^+$03] and RJoin [ILK08] adopt a full indexing strategy where all database tuples are indexed in the network. In fact, RJoin generates $2 \times k$ indexing messages for each tuple of k attributes.

In P2P systems, several types of failures can occur preventing the system from producing any result. PIER and RJoin rely on the DHT to deal with crash failures of peers and thus does not give guarantees on result completeness. Fault tolerance is a widely-studied problem, but the environment of stream processing applications creates new challenges.

The use of a DHT for routing tuples (indexing) and as a hash table for storing tuples gives internet scalability. However, the indexing strategy adopted by the state of the art approaches is clearly not practical in stream environment where the amount of information is massive in volume, and the number of tables and attributes can be huge. Moreover, in these approaches node failures and problems generated by skewed data have not been addressed. In the next section we present the architecture of DHTJoin, a method for processing continuous join queries over distributed data streams.

CHAPTER 3

DHTJoin Architecture

In this Chapter we present the architecture and the core concepts of the DHTJoin method which is focused on meeting the challenges of efficiently executing continuous join queries in an network environment.

In DHTJoin, nodes are connected by means of an overlay network that provides a DHT service (Distributed Hash Table) [DZD+03]. The nodes from the overlay network are assigned unique identifiers. While there are significant implementation differences between DHTs [RFH+01][SMK+01], they all map a given key k onto a node p using a hash function and can lookup p efficiently, usually in $O(logn)$ routing hops where n is the number of nodes. DHTs typically provide two basic operations : $put(k, data)$ stores a key k and its associated $data$ in the DHT using some hash function ; $get(k)$ retrieves the data associated with k in the DHT. The basic structure of a node participating of DHTJoin is depicted in Figure 3.1.

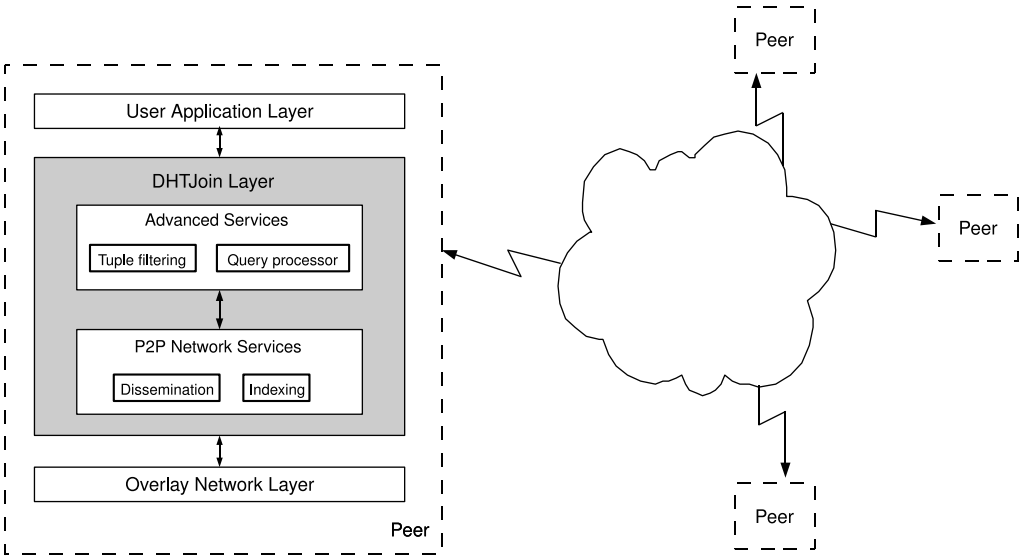


Figure 3.1 – Architecture of DHTJoin

The proposed architecture is based on three main components : (1) an interface for submitting the queries ; (2) a core layer named DHTJoin divided into two components : a component providing advances services as tuple filtering and query processing, and a component providing P2P network services as dissemination and indexing ; and (3) an overlay network layer that

provides a communication service to the DHTJoin layer. In DHTJoin, the user queries are expressed using the CQL language and they are parsed and disseminated to the overlay nodes. Data items (tuples) are filtered and indexed to nodes for later processing. Once a node receives a query, it instantiates the join operators that allow the processing of tuples stored locally. During execution, a node may produce answer tuples which are forwarded to the node submitting the query. Depending on the functionality required by a node, one or more components of the architecture may not exist. The different tasks performed by nodes during the execution of a query give raise to three kind of nodes. The first kind is Stream Reception Peers (SRP), a SRP filters the tuples received from data stream sources. Once a tuple is filtered it is indexed to the second kind of nodes, the Stream Query Peers (SQP). SQPs are responsible for executing query predicates over the tuples stored locally. Result tuples are sending the the third kind of nodes, the User Query Peers (UQP). Note that the difference between SRP, SQP and UQP is functional and the same node can support all these functionalities.

The rest of this chapter is organized as follows. In Section 3.1 we first present the DHTJoin stream data model. In Section 3.2, we describe issues concerning tuple filtering and query processing. In Section 3.3, we describe the various ways DHTJoin uses a DHT overlay network. Finally, in Section 3.4 we present the data flow during query processing.

3.1 Stream Data Model

In DHTJoin a data stream (a stream in the following) is an append-only sequence of data items called tuples, generated at data sources, which are composed of attribute values. DHTJoin does not maintain metadata, thus each tuple is self-describing. All tuples belonging to the same stream have the same set of attributes. This set of attributes defines the schema of the stream. In addition to application-specific attributes, each tuple has system-assigned attributes. This attributes are hidden from the application and are used internally by DHTJoin for query processing purposes. For example, upon arrival to the system every tuple is timestamped to indicate its arrival time, and every tuple generated (i.e. a join tuple) has the timestamp of the oldest tuple that was used in generating it. More specifically, DHTJoin defines a tuple, a stream , a data source and a relation as follows :

Definition 1. *A tuple is a data item on a stream taking the form :* $(timestamp, stream, a_1, ..., a_n)$*, where timestamp represents the time that they are inserted in the overlay network by some node, stream is the name of the stream which the tuple belongs to and* $a_1, ..., a_n$ *are attribute values. The schema has the form* $(TS, S, A_1, ..., A_m)$ *and all the tuples belonging to the same stream have the same schema.*

Definition 2. *A stream is an uniquely named unbounded bag (multiset) of tuples that all conform to the same schema.*

A stream originates at a single data source. However, a data source can produce multiples streams but it must assign a different name to each stream.

Definition 3. *A data source is any application or device that continuously produce tuples and pushes them to SRPs for processing.*

Definition 4. *A relation is time-varying bag of tuples. Note that this definition differs from the traditional one which has no built-in notion of time.*

We assume that data and query sources are equipped with well-synchronized clocks by using the public domain Network Time Protocol (NTP) designed to work over packet-switched and variable latency data networks and already tested in distributed DSMS [TcZ07]. Since a data stream is assumed to be unbounded or at least unknown length, tuples are stored in a main memory space reserved to query operators and they are maintained only for a limited time using a sliding window model.

3.2 Advanced Services

In this section we present the Advanced Services component of the DHTJoin layer. Basically, this layer performs query processing and tuple filtering tasks. The query processor, compiles the user's CQL query text into an internal query plan and implements join query operators when it is responsible for executing the local portion of a query. Additionally, the query processor contacts the P2P Network Services in order to disseminate a query. Given a continuous join query, its join predicates are used to filter the incoming tuples of a data stream. Thus, tuples that do not satisfy the join predicates of a given query are dropped as early as possible. In this work we focus in join predicates but in the same way a filter based on selection predicates can be implemented. In the following we present the process of filtering tuples and the details of the query processor.

3.2.1 Query Processor

When an UQP submits a CQL query, the first task of the query processor is query parsing. In DHTJoin, the CQL statement is parsed checking that it is correctly specified and subsequently it is converted into a query plan. Syntactically, CQL is a relative minor extension to SQL [AW04] that uses three classes of operators over streams and relations (see Section2.1.1). In this work we use a stream-to-relation operator that takes a stream as input and produces a relation as output. The stream-to-relation operator in CQL is based on the concept of sliding window. Specifically, DHTJoin queries are specified using time-based sliding windows in order to limit the size of the state maintained by a join operator. A time-based sliding window on a stream S takes a time interval W as a parameter and produces a relation as output. At time τ, the relation contains all tuples of S with timestamps between $\tau - W$ and τ. Syntactically, a time-based sliding windows is specified by following S with [Range W].

Once a query is parsed, a query plan is compiled from it. A query plan is composed of operators, which perform the processing of tuples, queues, which buffer tuples as they move between operators, and states, which store operator state. In a plan, a queue connects a "producing" plan operator O_P to its "consuming" operator O_C. A collection of tuples representing a portion of a data stream or relation is contained at any time in a queue. The elements produced by an O_P are inserted into the queue remaining there until they are retrieved by O_C as needed. An state has a behaviour depending of the operator it belongs to. For example, to perform a join of two streams using sliding windows, the join operator probes all tuples in the current window on each input stream maintaining one state for each of its inputs. A query plan can be thought of as a dataflow diagram that pipes table data through a graph of query operators. Performance requirements of data stream applications often dictate that buffers and queues must be kept in memory, and we make that assumption throughout this work.

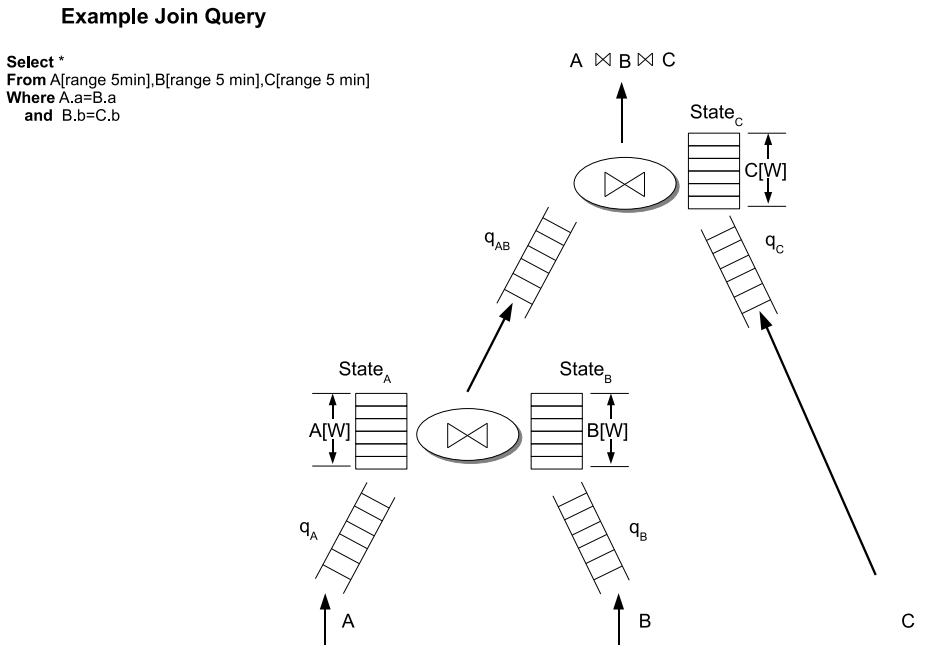


Figure 3.2 – A query plan illustrating operators, queues and states.

Locally, SQP nodes performs join processing using the MJoin operator (see Section2.1.2.2). The execution of a MJoin operator can be seen as a sequence of query plans executed using left-deep pipeline plans where the internal state of hash tables is determined solely by the source tuples that have arrived so far. As a example, a query plan of a join query, implementing the *probing sequence* $A \rightarrow B \rightarrow C$ for tuples arriving from stream A, is shown in Figure 3.2. There are two MJoin operators that perform the join processing using the tuples stored in the states. Queues q_A,q_B and q_C hold the tuples arriving from streams A, B and C respectively. Queue q_{AB} holds the $A \bowtie B$ tuples. Note that mjoin does not store intermediate results. $State_A$ holds tuples for "A[Range $5min$]". $State_B$ and $State_C$ hold tuples for "B[Range $5min$]" and "C[Range $5min$]" respectively. States are maintained by the MJoin operator using sliding windows and tuples obtained from the queues are used to perform joins with tuples on the opposite state. State maintenance is performed using the direct approach technique [GÖ03b]. For every new tuple stored into one of the join states, expiration is performed at the same time as the processing of the new tuple. A tuples expires if its timestamp falls out of the range W of the window. Additionally, state maintenance could be triggered when there are no arrivals for some time.

When an UQP submits a query, its is disseminated. In this case the query processor contacts the P2P Network Service in order to disseminate a query. When an SRP receives a query, it extracts the necessary information for constructing the data structure that supports the tuples filtering. In the same way, an SQP extract the information concerning the query plan in order to instantiates the operators, queues and states that support the query processing.

3.2.2 Tuple Filtering

Once stream data items (tuples) arrive on a SRP they are filtered. The filtering process drops tuples not concerned by queries as early as possible. Basically, an SRP extract the information contained in the join predicates in order to set the data structure supporting the filtering process. Given a tuple, a filter finds a join predicate that match. When an arriving tuple is not dropped it is sent to an SQP for processing.

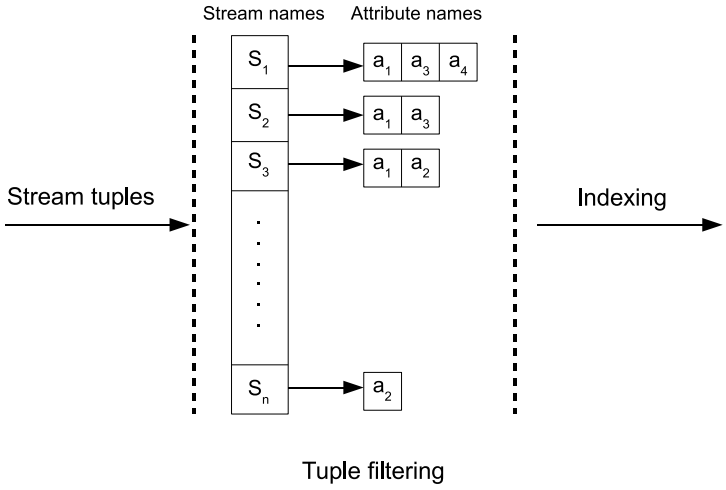


Figure 3.3 – Filtering tuples in DHTJoin

The filtering process can be accelerated by indexing the join predicates present in a given query. The indexing process is performed as follows. When an SRP receives a query, it extracts the information contained in the query plan concerning the stream names and the join attributes. The stream names are used as a key k mapping to a disjoint set of join attributes. As an example, when an SRP receives a query with a join predicate $S_1.a_1 = S_2.a_1$, it generates the keys S_1 and S_2, and adds the attribute a_1 to its corresponding set of attributes. Thus, each join predicate of the arriving queries is processed giving raise to a data structure similar that of Figure 3.3.

When a tuple arrives to an SRP, it extracts the name of the stream which the tuple belongs to and test it against the filter index. If there is an entry for the stream name in the filter index the SRP sends the tuple to an SQP for processing. Otherwise, the tuple is dropped. When an SRP send a tuple to an SQP it contacts the P2P Network Service. The filter index is organized as a hash table using the stream names as keys. Thus, the filtering process takes $O(1)$ in order to know if a tuple must be dropped.

3.3 P2P Network Services

The design of DHTJoin is based on Chord which is a simple and very popular DHT. However, DHTJoin is DHT agnostic and it can be adapted to others DHTs such as Pastry [RD01]

and Tapestry [ZHS+04]. Basically, DHTJoin uses a DHT in order to support two services : dissemination and indexing.

3.3.1 Dissemination

When an UQP submits a CQL query, it is parsed and a query plan is generated by the query processor (see Section3.2.1). Subsequently, the submitted query is disseminated to the participating nodes using the technique presented in [EAABH03]. The dissemination is based on a tree structure built using the information stored in the routing table maintained by the DHT. The basic idea is to consider that in a DHT as Chord a *lookup* operation can be perceived as a binary search that generates a binary tree using the nodes (links) stored in the routing table. The root of the tree is the node that submits the query (see Figure 3.4). When an UQP originates a query dissemination the query processor contacts the dissemination service, and each node receiving a dissemination message contacts its local query processor in order to announce the existence of a new query (see Figure 3.4).

The query is disseminated from the root node to all nodes of the DHT using a divide-and-conquer approach. Roughly, a node that receives a dissemination message $Dmsg$ store the query in a query table QT, creates a new $Dmsg$ and looks in its routing table to choose the nodes which to disseminate the query.

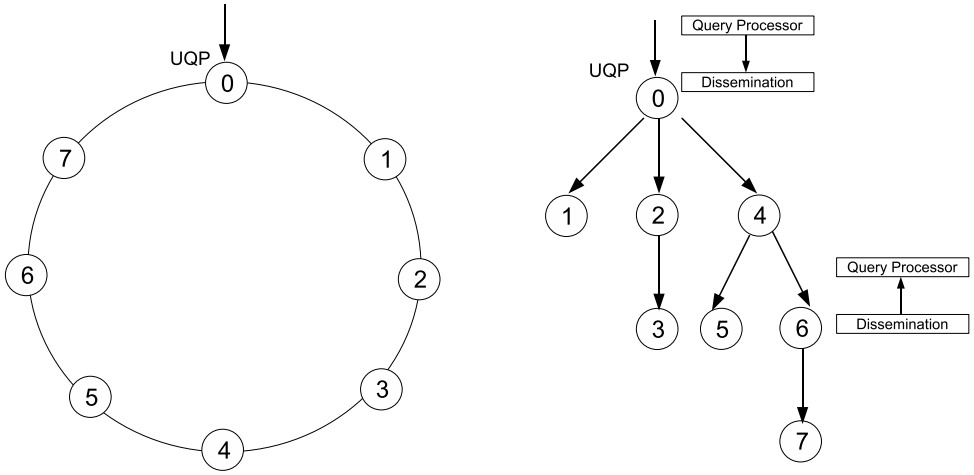


Figure 3.4 – Dissemination of a query.

The storage requirements of QT are the followings. The query identifier q_{id} : 2 bytes. The identity of UQP : 2 bytes. In the case of the query plan, we consider a query plan based on a bushy tree because it requires more storage space. Considering a bushy tree and a maximum of 10 streams and 10 attributes each, we need an array of size $2^{log_2^{10}} - 1 \approx 9$ to represent it. We can encode the streams and the attributes using 4 bits each. The root node, that needs to store more information, can be stored using 40 bits (5 bytes). Therefore, the query plan needs $9x5 = 45$ bytes. Considering a system with thousands of queries, the size of QT is $(2+2+45)*1000 = 49000$

bytes, i.e. approx. 48KB. We consider that this small structure can be stored in the RAM of DHTJoin nodes.

3.3.2 Indexing

Indexing a tuple amounts to storing an arriving tuple at one SQP of the overlay. A tuple is indexed by value using the attributes present in join predicates. Thus, when a tuple is not dropped by the filtering process, the filtering service contacts the indexing service (see Figure 3.1) in order to index the tuple. When an SQP receives a tuple via an indexing message, it is pushed to the local query processor where it enters a queue operator that contains tuples belonging to the same stream (see Figure 3.2) . We recall that we consider each tuple is self-describing, thus a tuple can reach easily the corresponding queue and subsequently gather join processing according to the query plan.

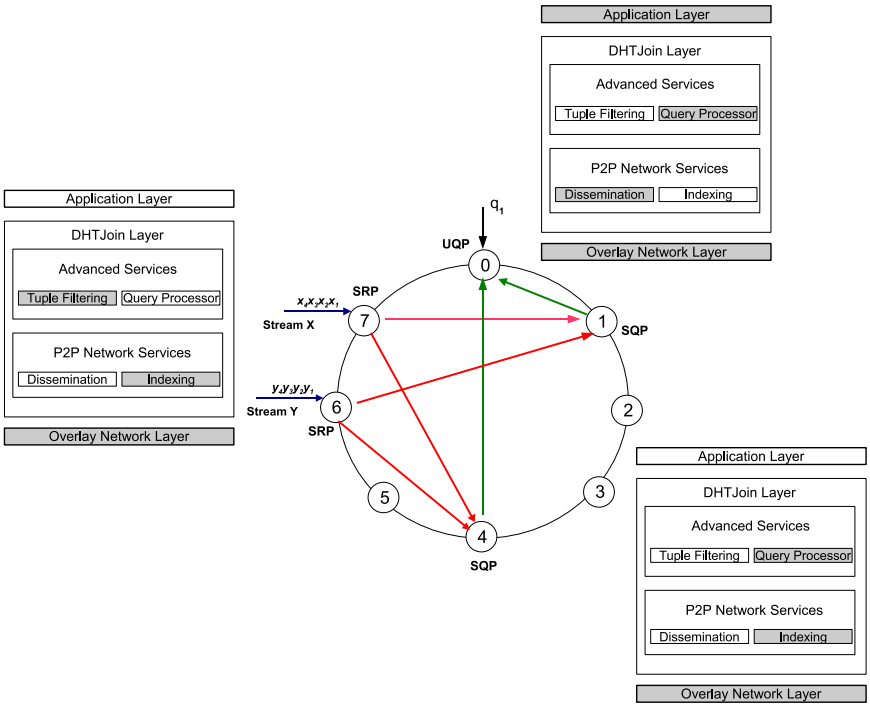


Figure 3.5 – Data flow in DHTJoin.

3.4 Data Flow

The data flow is the continuous flow of tuples and queries from tuples and query sources to client applications through the query processing nodes. To describe this flow we assume that the typical temporal order of events in DHTJoin is as follows. First, a CQL query is submitted and disseminated by an UQP. Then, tuples arrive to SRPs. The description of the data flow ignores the details of each component of the architecture and concentrates principally on the interactions between the different kind of nodes during stream processing. The different levels

of the architecture involved in the described interactions are represented by shadow areas in Figure 3.5.

When a UQP submits a query to the system, the application layer contacts the query processor in order to parse and generate a query plan. Subsequently, the query processor contacts the dissemination module of the P2P Network Service which in turn contacts the Overlay Network Layer that sends the dissemination message using the *put* primitive (see Figure 3.5). Tuples arriving at SRP nodes are filtered by the Tuple Filtering module of the Advanced Service. When a tuple is not dropped by the filter, it is indexed (stored) at one node of the DHT using the attributes involved (if any) in a previously disseminated query. To this end, the Indexing module of the P2P Network Service is used which in turn contacts the Overlay Network Layer that sends the indexing messages using the *put* primitive (see Figure 3.5). Finally, tuples indexed at SQPs are pushed to the local query processor where they are processed by the queries and result tuples generated at SQPs are sent back to the UQP.

3.5 Conclusion

In this chapter, we presented the architecture of DHTJoin to set the context of our work. We presented the DHTJoin stream data model, the different kinds of nodes involved in query processing, the different layers of the architecture and their principal modules, and the data flow during query processing. DHTJoin architecture is based on a DHT which influences many tasks that we enumerate as follows.

- Query Dissemination. The content of the routing table of each DHT node allows the construction of query dissemination trees rooted at the node submitting the query.
- Hash Index. When tuples are indexed into the DHT, the stream is stored in a distributed hash index keyed on the attribute(s) present(s) in the join predicates.
- Flow Partitioning and Parallelism. Arriving tuples belonging to different streams are partitioned by value using the DHT. This allows the distribution of join processing across the nodes belonging to the DHT giving rise to different approaches for parallelization of queries.

In the next chapter, we precise the type of join queries processed by DHTJoin, the approaches for parallelization of queries, the choice of the join operator, and how query plans are specified and optimized. We propose a theoretical approach in order to determine the number of computing resources needed to achieve a certain degree of completeness for a given query. Moreover, we describe how DHTJoin deal with problems as node failures and data skew.

CHAPTER 4

DHTJoin

In this chapter, we describe DHTJoin, our approach for processing continuous join queries over distributed data streams. In DHTJoin, nodes are organized using a DHT protocol. Basically, DHTJoin combines hash-based placement of tuples in a DHT and dissemination of queries by exploiting the embedded trees in the underlying DHT. Nodes insert data in the form of relational tuples and queries are represented in a relational query language for data streams such as CQL [AW04]. Tuples belonging to the same stream are inserted by the same node and continuous queries are originated at any node of the network. Tuples and queries are timestamped to represent the time that they are inserted in the network by some node.

Many applications are interested in making decisions over recently observed tuples of the streams. This is why we maintain each tuple only for a limited time. This leads to a sliding window $S[W_i]$ over S_i that is defined as follows. Let W_i denotes the size of $S[W_i]$ in terms of seconds, i.e. the maximum time that a tuple is maintained in $S[W_i]$. Let $TS(s)$ be a function that denotes the arrival time of a tuple s and t be current time. Then $S[W_i]$ is defined as $S[W_i] = \{s | s \in S_i \wedge (t - TS(s) \leq W_i\}$. Tuples continuously arrive at each instant and expire after W_i time steps (time units). Thus, the tuples under consideration change over time as new tuples get added and old tuples get deleted.

DHTJoin also deals with node failures during query execution and skewed data which may hurt load balancing and result completeness.

The rest of this chapter is organized as follows. In Section 4.1 we define formally the type of continuous join queries tackled in our approach. Section 4.2 presents the problem definition. In Section 4.3 we describe DHTJoin. Section 4.4 describes how DHTJoin deals with node failures. In Section 4.5, we provide an analysis of result completeness of our method which relates memory constraints, stream arrival rates and results completeness. Finally, Section 4.6 describes how DHTJoin deals with data skew.

4.1 Continuous Join Queries

Let us now formally define continuous join queries and the type of continuous queries that we consider in our approach. Let $S = \{S_1, S_2, ..., S_m\}$ be a set of data streams. Each data stream S_i has a relational schema $(A_1^i, A_2^i, ..., A_{n_i}^i)$, where each A_j^i is an attribute. We use equi-join and conjunctive predicates, i.e., the *where* clause uses exclusively conjunctions of atomic equality conditions. Let $Q_i = (S', \mathcal{P})$ be a continuous join query defined over $S' \subseteq S$ and composed by $\mathcal{P}$ that represents a set of equi-join predicates. As in [ZYYZ06][ILK08], we identify two types of join queries depending on the attributes involved in $\mathcal{P}$. A query of type 1 is a join query with a set of equi-join predicates as following : $\mathcal{P} = \{(S_1.A_k^1 = S_2.A_k^2), (S_2.A_k^2 = S_3.A_k^3), ..., (S_{m-1}.A_k^{m-1} = S_m.A_k^m)\}$, i.e., the join attribute is the same in all the relations of the query. As an example, a query of type 1 with three streams X, Y and Z having the same schema (A, B, C) is as follows :

q_{type_1} : Select sum (X.size)
From X[range 10 min], Y[range 10 min], Z[range 10 min]
Where $X.A = Y.A = Z.A$

A query of type 2 is a join query with a set of equi-join predicates as following : $\mathcal{P} = \{(S_1.A_k^1 = S_2.A_k^2), (S_2.A_l^2 = S_3.A_l^3), (S_3.A_m^3 = S_4.A_m^4), ..., (S_{m-1}.A_{n_m}^{m-1} = S_m.A_{n_m}^m)\}$, i.e., the join attributes are different and adjacent joins must have a common relation. As an example, a query of type 2 over the same streams of query 1 is as follows :

q_{type_2} : Select $Y.B$, $Z.C$
From X[range 5 min], Y[range 5 min], Z[range 5 min]
Where $X.B = Y.B$ and $Y.C = Z.C$

Queries of type 1 are often founded in network management applications. As an example, an application that monitors the traffic that passes through three routers and has the same destination host within the last 10 minutes. Considering the three kind of nodes of the architecture of DHTJoin (see Chapter 3), the traffic of the three routers can feed three SRPs where the packets are filtered and indexed to SQPs. This monitoring task can be performed using a multi-way window join query. The CQL query representing the task is :

q_1 : Select sum (X.size)
From X[range 10 min], Y[range 10 min], Z[range 10 min]
Where $X.destIP = Y.destIP = Z.destIP$

where X, Y, Z represents the stream generated by the traffic of the three routers and $destIP$ is the IP address of the destination host. Since the monitoring task is interested in the most recent content of the streams (the last 10 minutes) a time-based sliding windows is specified in the From clause of the query.

Queries of type 2 are more complex queries that allow to show how DHTJoin executes and addresses more sophisticated query plans.

4.2 Problem Definition

We view a data stream as a sequence of tuples ordered by monotonically increasing timestamps. The nodes have an identifier denoted by $node_{id}$ and are assumed to synchronize their clocks using the public domain Network Time Protocol (NTP),thus achieving accuracies within milliseconds [BGGMM04]. Each tuple and query have a timestamp that may be either implicit, i.e. generated by the system at arrival time, or explicit, i.e. inserted by the source at creation time.

This work focuses on query execution (not query optimization). Thus, we assume the existence of a query optimizer that translates a query represented in CQL [AW04] into a query plan in the form of an operator tree. Since an MJoin operator [VNB03] is used by default to specify join operations, only the join order needs to be specified by the optimizer, i.e. the choice of how to execute MJoin operators (e.g. which nodes) is done at runtime using our method. Each query Q_i has a query plan Q_{plan_i} that specifies the ordering of the join operations.

Formally, the problem can be defined as follows. Let $S = \{S_1, S_2, ..., S_m\}$ be a set of data streams, and $QP = \{Q_{plan_1}, Q_{plan_2}, ..., Q_{plan_n}\}$ be a set of query plans of the following set of continuous join queries $Q = \{Q_1, Q_2, ..., Q_n\}$, where $Q_i = (S', \mathcal{P})$ is a continuous join query defined over $S' \subseteq S$ and $\mathcal{P}$ represents a set of equijoin predicates. Our goal is to provide an efficient method to execute QP over S in terms of network traffic.

Additionally, we precisely state the main issues that our solution must tackle in the context of processing continuous join queries in DHTs.

- Node failures : in DHT networks nodes join and leave the system at will. In the context of join query processing, a failing node can lead to the waste of resources involved in sending, processing and storing tuples.
- Load balancing : hash-based placement of tuples is vulnerable to the presence of skew in the underlying data. This situation leads to load imbalancing that hurts any of the gains due to parallelism.
- Result completeness : the notion of completeness, defined as the ratio of the amount of data of the answer w.r.t. a query and the amount of answers we would get if all the participant nodes would respond, is very important in data stream applications since approximate answers are often sufficient when the goal of a query (i.e. a join query) is to understand trends.

4.3 DHTJoin Method

In this section, we describe DHTJoin, a method for processing continuous join query processing using DHTs. Basically, DHTJoin has two steps : dissemination of queries and indexing of tuples. A query is disseminated using the embedded tree inherent to DHTs networks and a tuple inserted by a node is indexed, i.e., stored at another node using DHT primitives. However, a node indexes a tuple only if there is a query that contains an attribute of the arriving tuple in $\mathcal{P}$. To this end, a node stores locally a disseminated query and once it receives a tuple it checks for already disseminated queries that contain an attribute of the arriving tuple in $\mathcal{P}$.

We describe the design of DHTJoin based on Chord which is a simple and very popular DHT. However, the techniques used here can be adaptable to others DHTs such as Pastry [RD01] and Tapestry [ZHS+04].

As we state in Chapter 3, in order to process a query, we consider three kinds of nodes. The first kind is Stream Reception Peers (SRP) for indexing tuples to the second kind of nodes, the Stream Query Peers (SQP). In Figure 4.1, nodes 3, 6 and 7 correspond to SRP because they receive tuples belonging to streams z, y, and x respectively. SQP are responsible for executing query predicates over the arriving tuples using their local sliding windows, and sending the results to the third kind of node(s), the User Query Peers (UQP). In Figure 4.1, nodes 1 and 4 are SQP because node 1 computes the join predicate $X.B = Y.B$ of query q_2 (submitted at node 0) and node 4 performs the join predicate $Y.C = Z.C$ of q_2. In addition, node 0 is a UQP

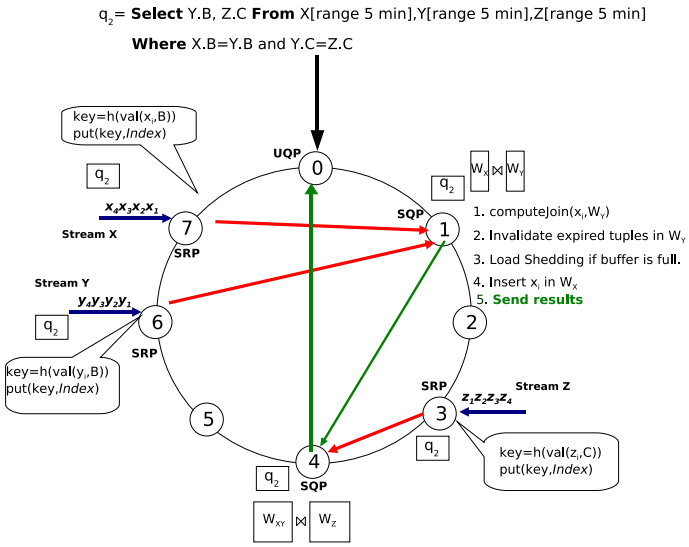


Figure 4.1 – A DHTJoin example using a query type 2

because query q_2 was submitted at this node.

Note that the difference between SRP, SQP and UQP is functional and the same node can support all these functionalities.

4.3.1 Disseminating Queries

Each new query issued by users should be disseminated to all nodes because by using S' and the set of predicates $\mathcal{P}$ of a query a node decide which tuples and attributes should be indexed. The query dissemination system consists of a set of DHT nodes. A query can originate at any of the nodes and is disseminated using a tree [CJK+03]. Basically, the query is disseminated from the root node to all nodes of the DHT using a divide-and-conquer approach.

To disseminate a query, DHTJoin dynamically builds a dissemination tree as proposed in [EAABH03]. The basic idea is to consider that in a DHT as Chord a *lookup* operation can be perceived as a binary search [EAABH03] that generates a binary tree using the nodes (links) stored in the routing table. The root of the tree is the node that submits the query (an UQP). This node sends the query to all its neighbors stored in its *Finger* table. If the *Finger* table contains redundant fingers only the last one is used and the others are skipped (see Algorithm 8). The forwarding space of each receiving nodes is restricted by a *Limit*. As shown in the Algorithm 8, the *Limit* for a node stored in the i^{th} position of the *Finger* table is that of $Finger[i+1]$, $1 \leq i \leq M-1$ where M is the size of the *Finger* table. In the case of the M^{th} finger, the *Limit* is set to the $node_{id}$.

When a node receives a disseminated query, it is stored locally in a *query table* (QT), thus allowing to know what is the attribute of an arriving tuple that must be used in the indexing process. This is important since a tuple s_i is indexed using an attribute A^i_j only if it is contained in the set $\mathcal{P}$ allowing to decrease network traffic and providing a better utilization of local SQP resources by avoiding the indexing of tuples using an attribute that is not being involved in a

Algorithm 8 InitDissemination : An UQP submits a query to the system. The UQP is the root of a dissemination tree and disseminates the query to all the nodes stored in its Finger table.

Require: **Q**, a join query. $\mathbf{Q_{plan}}$, the query plan.

1: **for** $i = 1$ to $M - 1$ **do**
2: **if** $(Finger[i] \neq Finger[i+1])$ **then**
3: $Receiver \leftarrow Finger[i]$
4: $Limit \leftarrow Finger[i+1]$
5: $Dmsg \leftarrow \langle node_{id}, q_{id}, Q, Q_{plan}, ts, Limit \rangle$
6: **send** $Dsmg$ to $Receiver$
7: **end if**
8: **end for**
9: $Receiver \leftarrow Finger[M]$
10: $Dmsg \leftarrow \langle node_{id}, q_{id}, Q, Q_{plan}, ts, node_{id} \rangle$
11: **send** $Dsmg$ to $Receiver$

query.

To disseminate a query, an UQP node creates a dissemination message $Dmsg$ containing its own node identifier $node_{id}$, an unique query identifier q_{id}, the query $Q = (S', \mathcal{P})$, the query plan Q_{plan}, a timestamp ts that denotes the arrival time of Q_i and a limit of dissemination $Limit$. A node with identifier $node_{id}$ that receives a $Dmsg$ store the query in its QT and forwards it to nodes stored in its $Finger$ table with identifiers belonging to the interval $]node_{id}, Limit[$. Subsequently, it creates a new $Dmsg$ preserving the $node_{id}$, the q_{id}, the Q, the $Qplan$, the timestamp ts, and changing $Limit$ by $NewLimit$ (see Algorithm 9). Note that the $NewLimit$ is used to define a new smaller subtree.

Algorithm 9 Dissemination : A node receives and forwards a dissemination message.

Require: **Dmsg**, a dissemination message.

1: **for** $i = 1$ to $M - 1$ **do**
2: **if** $(Finger[i] \neq Finger[i+1])$ **then**
3: **if** $(Finger[i] \in]node_{id}, Limit[)$ **then**
4: $Receiver \leftarrow Finger[i]$
5: **if** $(Finger[i+1] \in]node_{id}, Limit[)$ **then**
6: $NewLimit \leftarrow Finger[i+1]$
7: **else**
8: $NewLimit \leftarrow Limit$
9: **end if**
10: $Dmsg \leftarrow \langle node_{id}, q_{id}, Q, Q_{plan}, ts, NewLimit \rangle$
11: **send** $Dsmg$ to $Receiver$
12: **else**
13: $exit for$
14: **end if**
15: **end if**
16: **end for**

For example, using a fully-populated Chord ring with 8 nodes, each one contains a routing

table of $log(n)$ entries called fingers. The i^{th} entry in the table at node n contains the identity of the first node that succeeds or equal $n + 2^i$. A dissemination message initiated at node 0 is sent to finger nodes 1, 2 and 4 (see Figure 4.2) giving them the disseminations limits [1,2), [2,4) and [4,0) respectively. The disseminations limits are used to restrict the forwarding space of a node and they are constructed using as a upper bound the finger $i + 1$. Each node applies the same principle reducing the search scope. When node 2 receives the dissemination message with limits [2,4) it examines the routing table and sends the message to node 3. Once node 4 receives the dissemination message it examines the routing table and sends the message to nodes 5 and 6 with limits [5,6) and [6,0) respectively. In the same way, node 5 does not continues with the dissemination process (since there are no nodes between [5,6)) and node 6 disseminates the message to node 7. This forwarding process generates $n-1$ messages and a tree of depth $log(n)$,

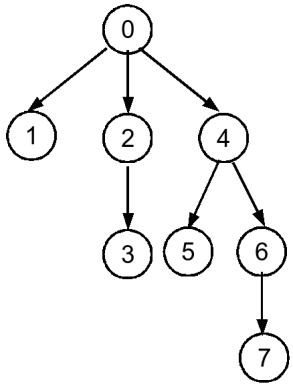


Figure 4.2 – A dissemination tree formed using DHT links of a 8-node Chord ring

which fixes the latency of query dissemination.

4.3.2 Indexing Tuples

The indexing of tuples allows DHTJoin to distribute the query workload across multiple DHT nodes. Let us describe how DHTJoin indexes tuples for streams $S = \{S_1, S_2, ..., S_m\}$. Let s_i be a tuple belonging to S_i. Let $A = (A_1^i, A_2^i, ..., A_{n_i}^i)$ be the set of attributes in s_i and $val(s_i, A_j^i)$ be a function that returns the value of the attribute $A_j^i \in A$ in tuple s_i. Let h be a uniform hash function that hashes $val(s_i, A_j^i)$ into a DHT key, i.e. a number which can be mapped to a $node_{id}$. A SRP that index a tuple $s_i \in S_i$ creates a message $Index = (S_i, s_i, A_j^i, ts)$ containing the stream S_i which the tuple belongs to, the tuple s_i being indexed, the attribute used to index the tuple and a timestamp ts that denotes the arrival time of the tuple. Let $S[W_i]$ denote a sliding window on stream S_i. Recall that we use time-based sliding windows where W_i is the size of the window in time units. At time t, a tuple s_i belongs to $S[W_i]$ if it has arrived in the time interval $[t - W_i, t]$.

For indexing a tuple s_i that arrives at an SRP, each tuple obtains an index key computed as $key = h(val(s_i, A_j^i))$. The attributes A_j^i in s_i are chosen once the tuple pass through the Tuple filtering module (see Section 3.2.2). Recall that the filtering process drops tuples not concerned by queries as early as possible. Then to index s_i the SRP node creates a $Index$ message and sends it to an SQP, by performing $put(key, Index)$. Thus, tuples of different streams having the same key are put in the same SQP node and are stored in sliding windows where they are

processed to produce the result of a specific join predicate.

4.3.3 Query Execution

Query processing in a DSMS entails the generation and execution of a query plan. This paper focuses on the execution part. For simplicity, we assume that the query plan is an operator tree that specifies the ordering of operations (i.e. join order) and it is included in the *Dmsg* message of the query dissemination step (see Section 4.3.1).

Queries of type 1 are executed using partitioned parallelism [Has95] with SQP nodes implementing the MJoin operator [VNB03]. In partitioned parallelism, individual operators are parallelized by partitioning their input across a network of nodes. The partial outputs from individual operators are unioned to form the final output result. This method is very common for processing queries with large inputs in a distributed system. The avantage of partitioned parallelism is that it gives more scalability for unbounded inputs because individual operators are easily partitioned into many independent units. A query plan contains a *probing sequence* for each stream present in the query (see Figure 4.3) that could be optimized locally, thus generating a new operator tree. Each node in the operator tree represents a join operator and an edge represents the next stream to probe. As we stated before, the partitioned parallelism gives more scalability but its combination with the MJoin operator gives more flexibility face to variations in the workload because the shape of the query plan residing on individual machines is restructured easy and independently.

Queries of type 2 are executed using pipelined parallelism [LR05]. In pipelined parallelism, a sequence of non-blocking operators is divided into smaller subsequences allocated to different nodes. This give raise to a execution scenario where nodes are connected via the producer-consumer relationship, where the output of the producer comprises the input of the consumer. In DHTJoin, queries of type 2 are executed using a segmented bushy processing strategy [LR05]. A segmented bushy tree applies independent parallelism with minimal dependencies among subtrees. We choose this strategy because it aims to balance partitioned and pipelined parallelism for complex multi-join queries giving more opportunities to share executions plans with queries of type 1. For queries of type 2, the query plan is assumed to be generated by a centralized query optimizer based on a cost model which captures information regarding data (e.g. tuples' arrival rates) and operators (e.g. cost of a join) [ZYYZ06]. Each node in the operator tree represents a join operator implemented using MJoin and an edge represents the next step in the pipeline.

In this section, we describe the execution of queries of type 1 and 2 in DHTJoin.

4.3.3.1 Queries of Type 1

In this type of queries, DHTJoin uses partitioned parallelism where different nodes execute independently the same query plan on different data partitions. By default, DHTJoin instantiates an MJoin operator [VNB03] for queries of type 1. Figure 4.3 shows an MJoin operator instantiated at a SRP for a 3-way continuous join query expressed using CQL [AW04]. There are three hash tables corresponding to the three join attributes of the query and three probing sequences. An MJoin operator is ready to accept a new tuple on any input stream at any time. MJoin implements a lightweight tuple router that, considering the *probing sequence*, routes the arriving tuples to the remaining hash tables.

MJoin integrates sliding windows as follows. Let us consider the MJoin operator of Figure 4.3. Each hash table stores the tuples that fall within the current window period which, in the

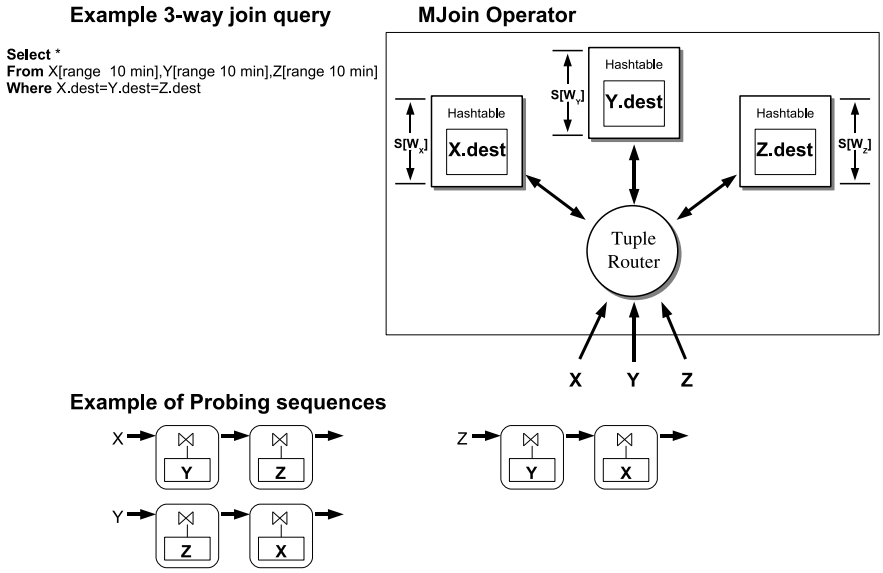


Figure 4.3 – MJoin operator for a 3-way join query of type 1

case of q_1,correspond to 10 minutes for the streams X, Y and Z. For example, for each arriving tuple $x \in X$, before probing the hash table on Y, the Y tuples that are outside of the window $S[W_Y]$ are eliminated. If an intermediate result tuple xy_i is generated, the Z tuples that are outside of the window $S[W_Z]$ are eliminated before the probe step.

Choosing a *probing sequence* is very important in MJoin because it must ensure that the smallest number of intermediate results is generated. This process is supported by heuristic-based ordering algorithms [GÖ03b][VNB03]. MJoin is very attractive when processing continuous queries over data streams because the query plans can be changed by simply changing the *probing sequence*. Thus, each SQP node that processes a query of type 1 can optimize the execution plan of the query independently.

Let us illustrate how DHTJoin performs query processing with the following query of type 1 :

q_1 : Select sum (X.size)
From X[range 10 min], Y[range 10 min], Z[range 10 min]
Where $X.dest = Y.dest = Z.dest$

As shown in Figure 4.4, the query q_1 is submitted at node 0 and disseminated, using the strategy proposed in Section 4.3.1, over the entire network as soon as it is submitted. SRPs 7, 6 and 3 filter x_i, y_i and z_i tuples, indexing them only if q_1 contains in its query plan an attribute belonging to the arriving tuples. Recall that in a query of type 1, the join attribute is the same in all relations, so that all the tuples having the same attribute value are located in the same

SQP without producing intermediate results. Therefore, q_1 can be executed independently at different SQPs. Thus, SQPs 1 and 4 process q_1 on different partitions of X, Y and Z.

As an example, the following steps are executed at SQP 1 (see Figure 4.4) when a new tuple $x \in X$ arrives :

- The *probing sequence* for an X tuple is $Y \rightarrow Z$ (see Figure 4.3).
- x is used to probe the hash table on Y to find the tuples that satisfy the join predicate $X.dest = Y.dest$. Intermediate tuples are generated by concatenating x with the matching tuples $y_i \in Y$, if any.
- If any result tuples were generated, they are routed to the Z hash table in order to find the tuples that satisfy the join predicate $Y.dest = Z.dest$.
- Before each probing step tuples that are outside of the windows $S[W_Y]$ and $S[W_Z]$ are eliminated.
- x is inserted into the X hash table.

The results produced by SQPs 1 and 4 are sent directly to the UQP (whose address was provided in the $Dmsg$ message when q_1 was disseminated).

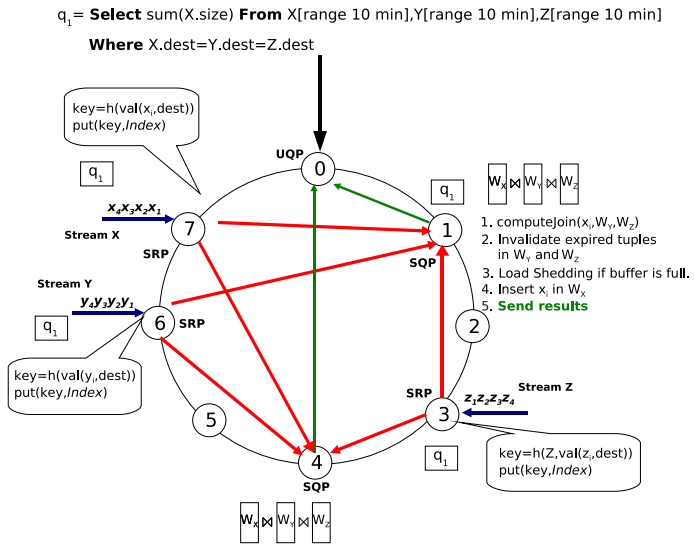


Figure 4.4 – Execution of q_1 in DHTJoin

4.3.3.2 Queries of Type 2

DHTJoin executes queries of type 2 using pipelined parallelism [LR05] where different nodes run in a pipelined fashion such that tuples output by a node can be fed to another node as they get produced. Recall that DHTJoin partitions the streams by hash functions. For example, let us consider query q_2 with the following set of predicates $\{(X.B = Y.B), (Y.C = Z.C)\}$. Streams X and Y are indexed based on the value of attribute B while stream Z is indexed based on the value of attribute C which is placed at a node different from where the stream Y is indexed. Therefore, redirection of intermediate join results is necessary in this type of query. Another

solution is to index the stream Y twice, i.e. based on attributes B and C executing $X \bowtie_B Y$ and $Y \bowtie_C Z$ in parallel. However, we do not consider this solution for the two following reasons :

- It duplicates unnecessarily the indexing of Y tuples.
- It introduces more messages and processing costs because the output tuples of the two joins must be processed to find the final join result.

For queries of type 2, we assume that the query optimizer generates a query plan based on a bushy tree of binary joins that has the potential of executing independents subtrees concurrently. Local operators are executed using an MJoin operator and can be optimized as for queries of type 1.

Let us illustrate how DHTJoin performs query processing using the following query of type 2 :

q_2 : Select Y.B, Z.C
From X[range 5 min], Y[range 5 min], Z[range 5 min]
Where X.B=Y.B and Y.C=Z.C

This query specifies an equijoin among X, Y and Z streams over the last 5 minutes. Query q_2 is submitted at node 0 and disseminated over the entire network as soon as it is submitted. Thus, after a while, all nodes know the existence of this query and are able to index the incoming streams (tuples). We assume that the query plan generated for q_2 is $(X \bowtie_B Y) \bowtie_C Z)$. Once an X-, Y- or Z-tuple arrives at nodes 7, 6 and 3 respectively, each node filter and subsequently index tuples concerned by q_2. If so, nodes 7, 6 and 3 execute the task of an SRP. For instance, in our example, node 7 indexes x_i because the attribute $B \in X$ is in the Q_{plan} of q_2. Node 7 creates a message $Index = (X, x_i, B, ts)$, generates an index key using $key = h(val(x_i, B))$ and indexes the tuple using $put(key, Index)$. The equijoin predicate $X.B = Y.B$ belonging to q_2 is evaluated at a SQP (node 1) only with tuples that arrive in the system after the query.

Sliding windows are used at each SQP node, as for queries of type 1, as follows. For example, at node 1 in Figure 4.1, tuples expired in $S[W_Y]$ are invalidated upon the arrival of X-tuples. The load shedding procedure is executed over $S[W_X]$'s buffer if there is not enough memory space to insert the arriving tuple.

The SQP node 1 searches in the query plan of q_2 what is the next step to follow and concludes that the intermediate results $x_i y_j$ must be sent to another node using the value of C attribute belonging to the Y-tuple. Thus the SQP node 1 creates a message $Index = (XY, x_i y_j, C, max(TS(x_i), TS(y_j))$, generates an index key using $key = h(val(y_j, C))$ and index the intermediate tuple using $put(key, Index)$ to SQP node 4. The join result tuples produced by SQP node 4 are immediately sent to the appropriate UQP node (whose address is provided when starting query dissemination).

4.3.4 DHTJoin on Other DHTs

The design of DHTJoin is based on Chord which is a simple and very popular DHT. However, the dissemination technique and the indexing of tuples used can be adapted to others DHTs such as Pastry [RD01] and Tapestry [ZHS+04]. In the dissemination of queries, recall that the basic idea (using Chord) is to consider a *lookup* operation as a binary search in spite of its ring geometry. The routing algorithms in Pastry and Tapestry are both similar in spirit to the PRR's routing algorithm [PRR99] which is based on a tree hierarchical organization. This makes of Pastry and Tapestry a good choice to implement the query dissemination.

We consider Pastry as an example and demonstrate how to apply query dissemination using the mechanism proposed in [CJK+03]. In Pastry each node has a unique *nodeId* assigned from a identifier space of 128 bits. Application-specific objects are assigned unique identifiers called keys from the same identifier space. Assuming a network of size n, each Pastry node maintains a routing table of $log_{2^b}n$ rows with 2^b entries each. For the purpose of routing, the *nodeId* and keys can be thought of as a sequence of L digits in base 2^b. The mechanism to route a message is prefix-based, i.e. the routing is achieved by forwarding the message to a node that shares a common prefix by at least one more digit. Pastry can route a message to any node in $log_{2^b}n$ hops. For ease of explanation, we use $b = 1$, $L = 3$ and a network of 8 nodes. A dissemination message initiated a node 000 contains the query id q_{id} and the message is sent to the 3 nodes of its routing table 100, 010 and 001 adding the routing table row r of each node. When a node receives a dissemination message, it searches in the routing table all the nodes located in rows greater than r (if any) and disseminates the message to them. This process is repeated at each node that receives the message, thus generating a dissemination tree of depth $log(n)$.

Regarding the indexing of tuples, we use primitives which represent capabilities that are common to all DHTs. DHTs typically provide two basic operations [DZD+03] : *put(k, data)* stores a key k and its associated *data* in the DHT using some hash function ; *get(k)* retrieves the data associated with k in the DHT. Thus, we can process continuous join queries in other DHTs.

4.4 Dealing with Node Failures

In this section, we discuss how DHTJoin deals with node failures during query execution. By node failure, we mean various situations by which a DHT node stops participating in query execution (e.g. because it crashes). We address this issue considering two situations : (1) *Failure of a node during query dissemination.* Recall that the dissemination of queries allows to decrease network traffic by avoiding the indexing of tuples using an attribute that is not being involved in a query. However, its benefits can be lost when the tree hierarchical organization of the dissemination is broken due to node failures. (2) *Failure of a node during query execution.* The failure of a node stops the indexing of tuples. With queries of type 2, this situation can generate partial results that never contribute to generate join results.

4.4.1 Failures during Query Dissemination

In DHTJoin, continuous join queries are originated at any node of the DHT and disseminated using a tree. The dissemination of queries achieves a network coverage of 100%, takes $O(logn)$ hops to reach every node in the network and generates $n-1$ messages. However, dynamic changes of the structure of the DHT network can disturb the dissemination. The failure of a node in the tree structure generated by the dissemination makes the entire subtree under this node unreachable. To provide reliability in the dissemination of queries, we propose to use a gossip based protocol as a complementary to our tree based dissemination. Thus, the dissemination algorithm, which guarantees to never send redundant messages, will attempt to disseminates queries, while the gossip protocol ensures that all nodes with high probability and low cost get the dissemination message.

Gossip algorithms mimic rumor mongering in real life. Just as people pass on a rumor by gossiping their contacts, each node in a distributed system relays new information it has received

to selected nodes which in their turn, forward the information to other peers, and so on. They are also known as *epidemic protocols* in reference to virus spreading. Basically, gossip proceeds as follows : a node n_i knows a group of other nodes or *contacts*, which are maintained in a list called $n_i's$ *view*. Periodically n_i selects a contact n_j from its view to gossip : n_i sends its information to n_j and receives back other information from n_j.

Gossip has recently received considerable attention from researchers in the field of P2P systems [VvS07][KvS07]. In addition to their inherent scalability, they are simple to implement, robust and resilient to failures. They are designed to deal with continuous changes in the system, while they exhibit reliability despite peer failures and message loss. This makes them ideally suited for large-scale an dynamic environments like P2P systems.

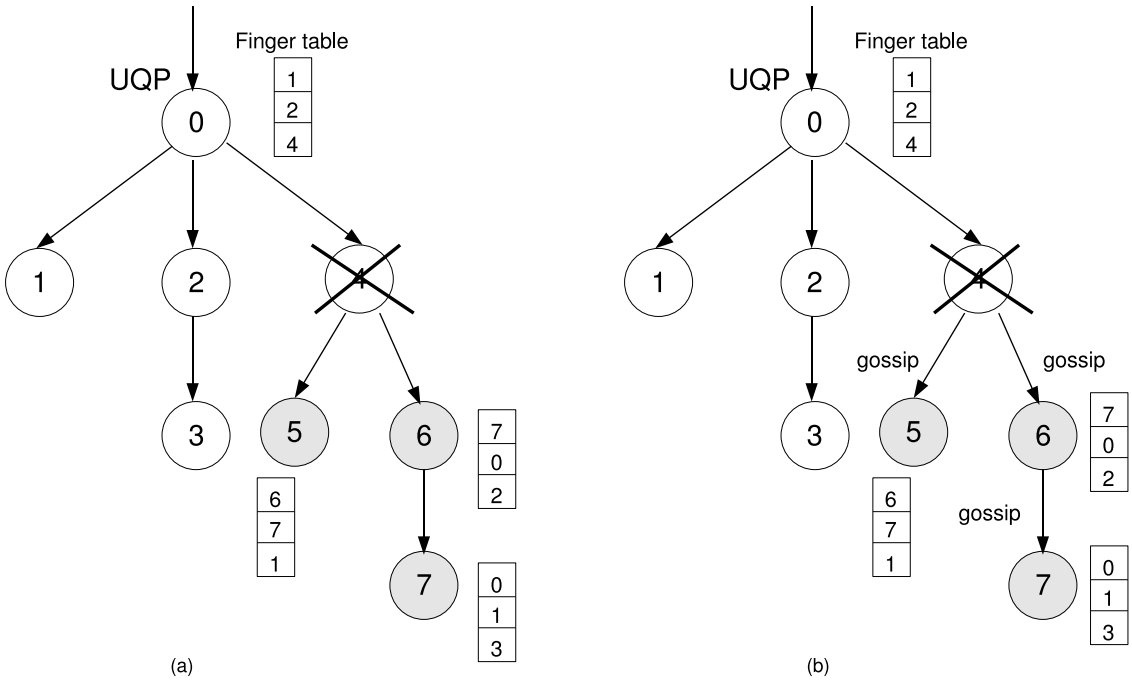


Figure 4.5 – Gossip and its integration with dissemination

The generic gossip behaviour of each node can be modeled by means of two separate threads exchanging information that depends strictly on the application : an active thread which takes the initiative to communication, and a passive thread which reacts to incoming initiatives [KvS07]. In a gossip protocol the information exchange can be implemented using the following strategies :

- Eager push approach : nodes send messages to random selected peers as soon as they receive it for the first time.
- Pull approach : periodically, nodes query random selected peers for messages recently received. If a node receives information about a message not received yet, it updates its local information.
- Lazy push approach : nodes send the message identifier to random selected peers as soon as they receive it for the first time. The peers receiving an identifier of a message they

have not received make an explicit pull request.

We propose a gossip protocol based on the concept of anti-entropy. The term anti-entropy [BHÖ+99] refers to a protocol that detects and corrects inconsistencies in a system by continuous gossiping. Basically, the gossip algorithm is deployed on top of the DHT using the information stored in the *Finger* table of nodes for random gossip exchange. We reason that imposing some level of determinism on the choice of which nodes to gossip reduce redundant messages while achieving complete dissemination under node failures. In general, a gossip anti-entropy algorithm progresses periodically through rounds (with a gossip period noted T_{gossip}) in which a node n_i randomly chooses other nodes to which it sends a message. The nature of the message depends of the approach chosen for the information exchange. When a node discovers that it has missed a message, it updates its local information. Our gossip protocol is based on a pull approach that periodically queries random selected peers for missed messages. We justify our decision on the following facts : (a) We have verified experimentally that in spite of the failures of the nodes the dissemination process provides a good network coverage. (b) Recent research [FKL+09] has showed that the pull approach is more efficient that a push approach when the information disseminated has covered a significant number of nodes. The problem of a pull approach is the setting of the gossip frequence T_{gossip} since a node that has received regularly all the disseminated messages does not have to start a gossip exchange. Thus, T_{gossip} must be particularly adapted to each node. We use the following steps to adapt T_{gossip} : (1) once arriving to the system, each node starts a gossip round, (2) a local query arrival rate is calculated by each node as queries arrive, (3) if after a time interval, equal to the local query arrival rate calculated in the previous step, the node does not receive a query then a gossip round is triggered.

As an example, Figure 4.5(a) shows the dissemination of a query using a dissemination tree rooted at node 0. During the dissemination process node 4 fails making the entire subtree under this node unreachable (see dashed nodes). Nodes 5, 6 and 7 run a gossip algorithm and pick up randomly a node from its *Finger* table which they send a message. If a node discovers that it has missed a query, then it updates its local state. For example, in Figure 4.5(b), the node 7 pick ups the id of node 3 from its *Finger* table. Node 7 that has not received the query, since node 4 has failed, updates its local information with the message received from node 3.

To support gossip, each node locally manages the following elements :

- $fingerTable$, this table correspond to the view maintained by gossip nodes. Basically, the gossip protocol extract from it the address IP of a node.
- QT, is the query table stored at each node in DHTJoin. Since the gossip protocol is used to detect and correct missed queries due to node failures, this table is used to detect if a gossip message contributes to the knowledge of query missed by a node.

The gossip behaviour of each node is illustrated in Algorithm 10 : the active behaviour describes how a node n_i initiates a gossip exchange, while the passive behaviour shows how a node n_j reacts to a gossip exchange initiated by n_i.

All the nodes that do not receive a query after a time interval T_{gossip} trigger a gossip round. Thus, a node n_i selects from its $fingerTable$ a node n_j via **select**() and then send to it $gossipMsg$, a message that contains the more recent queries received. n_i receives in exchange $gossipMsg'$ containing similar information from n_j. However, this messages is sent only if it contributes to detect queries missed by n_i. The procedure **merge**() collects in a $buffer$ all the entries from both the local QT and the recently received $gossipMsg'$, and discards the duplicates. Then, the QT and T_{gossip} are updated.

The passive behaviour is triggered when n_j receives a gossip message containing the most

Algorithm 10 Gossip behaviour of nodes

```
1: // active behaviour
2: while () do
3:    wait(T_gossip)
4:    n_j ← fingerTable.select()
5:    gossipMsg ← ⟨QT.getRecent()⟩
6:    send gossipMsg to n_j
7:    receive gossipMsg′ from n_j
8:    buffer ← merge(gossipMsg′, QT.getQueries())
9:    QT.update(buffer)
10:   update(T_gossip)
11: end while

12: // passive behaviour
13: while () do
14:   waitGossipMsg()
15:   receive gossipMsg from n_i
16:   buffer ← merge(gossipMsg, QT.getQueries())
17:   QT.update(buffer)
18:   update(T_gossip)
19:   if (missedQueries ← (gossipMsg ∩ QT) ≠ ∅) then
20:      gossipMsg′ ← missedQueries
21:      send gossipMsg′ to n_i
22:   end if
23: end while
```

recent queries received from some node n_i. Then, n_j updates its T_{gossip} and its local QT via **merge**() and **update**() as described previously, and sends back a gossip message only if it contributes to detect queries missed by n_i.

4.4.2 Failures during Query Execution

DHTJoin distributes the query workload across multiple DHT nodes and provides a mechanism that avoids indexing tuples using attributes not contained in the set $\mathcal{P}$ of a query. However, when a node fails, another node can generate partial results irrespective of whether they produce join query results. In this section, we address the problem of indexing partial results that never contribute to generate join results.

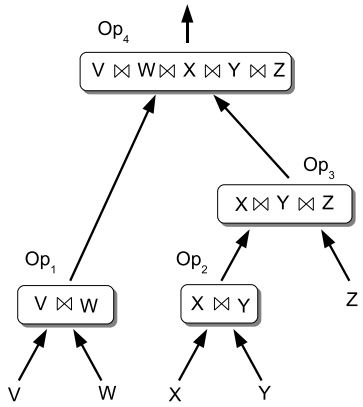


Figure 4.6 – Query Plan of a 5-way continuous join query of type 2

For example, let us consider the following query plan $(V \bowtie W) \bowtie ((X \bowtie Y) \bowtie Z)$ for a query of type 2 where there are nodes connected by a producer-consumer relationship, whereby a producer node generates tuples to be processed by a consumer node [YP08]. The query plan (see Figure 4.6) shows the relations between producers and consumers. We assume that a join operator Op_i resides at node n_i. Operator Op_3 is a producer of $X \bowtie Y \bowtie Z$ tuples for Op_4 and a consumer w.r.t Op_2 and SRP of stream Z. Recall that in a query of type 2, the operators are placed at different SQP nodes and the query plan is provided in the query dissemination step. If the node n_1 fails, the indexing of $V \bowtie W$ intermediate result tuples is stopped, thus yielding no join results because of no matching tuples in node n_4. Furthermore, if no matching tuple of $V \bowtie W$ appears at node n_4 before expiration of $X \bowtie Y \bowtie Z$ tuples, the resources involved in sending, processing and storing these tuples are wasted.

To address this problem, we propose the following solution. If node n_4, where Op_4 is executed, detects that $V \bowtie W$ tuples are not being generated by node n_1 it sends a message to node n_3 to alert that it is not necessary to send $X \bowtie Y \bowtie Z$ tuples. Consequently, as the demand of Op_3 as a consumer has changed, it propagates the alerting message to node n_2 and to the SRP of stream Z only if there does not exist another query that needs $X \bowtie Y \bowtie Z$ tuples generated by Op_3. This condition is verified at all the operators that receive an alerting message. Once the communication with n_1 is established again, node n_4 sends a resume message to n_3 in order to continue with the production of tuples and node n_3 propagates the resume message it proceeds.

If in a query plan, a consumer also acts as a producer, it is not necessary to alert its consumer. The reason is that a consumer is always testing its producers in the query plan in order to detect a problem. Therefore, the consumer that detects that there are tuples not being generated by a producer must trigger an alert message only to the other producers (the descendents) in the query plan if any. Procedure 11 describes the behaviour of the consumer that trigger the alert message to the producers of the query plan. Procedure 12 describes the behaviour of a producer in order to handle and alert message.

Procedure 11 Send_AlertMSG(q)

Require: the query q

1: **for** all the descendents $\in$ query plan of q **do**
2: $alertMSG \leftarrow \{q, \{suspend|resume\}\}$
3: send(myID,alertMSG)
4: **end for**

Procedure 12 Handle_AlertMSG(consumerID, alertMSG)

Require: **consumerID**, the identifier of the consumer node in the Chord ring. **alertMSG** is a message containing the identification of the query q and the type of action {suspend,resume}

1: **if** notExists($q_i \in$ QT $\neq q$) **then**
2: propagate_AlertMSG(myID, alertMSG) ;
3: **end if**
4: **if** (action is suspend) **then**
5: suspend(q)
6: **else**
7: resume(q)
8: **end if**

In Procedure 11, a consumer sends an alert message to all the other producers of the query plan of query q. The consumer sends a suspend message when it detects that there are tuples not being generated by a producer. Otherwise, it sends a resume message.

In Procedure 12, Line 1 verifies that there does not exist another query in QT that needs the tuples generated by the producer that receives the message. If so, the producer acting as a consumer sends the message to its descendents (Line 2) in the query plan of q. Finally, the producer performs appropriate operations to suspend (Line 5) or reactivate (Line 7) locally the production of tuples related to q. By eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic and a better utilization of local resources.

4.5 Analysis of Result Completeness

The notion of result completeness is important in distributed and P2P databases since partial (incomplete) query answers are often only possible [NFL04][KSH+08]. Result completeness is thus defined as the fraction of results actually produced over the total results (which could be produced under perfect conditions). In data streaming applications, the potential high arrival

rates of streams impose high processsing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurement or utilization patterns. Query approximation can be done by limiting the size of states maintained for queries [KNV03]. In our analysis we focus in the case where the memory allocated to maintain the state of a query is not sufficient to keep the window size entirely, thus reducing the received join results and completeness. DHTJoin provides more memory to store tuples, but we consider that determining the number of computing resources necessaries to achieve a certain degree of completeness for a given query is an important aspect in the setup phase of DHTJoin.

In this section, we propose formulas which relate peer memory constraints, stream arrival rates, and result completeness. We will use these formulas in our performance evaluation and they could be useful to a DHTJoin user (e.g. an application developer) to define and tune a DHT network for specific application requirements. We provide the necessary equations to calculate the completeness in a 2-way join and afterwards we generalize our results for a m-way join.

Table 4.1 – Symbols used in this analysis

Symbol	**Description**
n	number of nodes
m	number of streams
$S = \{S_1, S_2, ..., S_m\}$	set of streams
A_j^i	j-th attribute of stream S_i
λ_i	arrival rate of stream S_i in tuples/sec
$S[W_i]$	sliding window of stream S_i
W_i	window size of $S[W_i]$ in seconds
$Q = \{Q_1, Q_2, ..., Q_n\}$	set of queries
$\mathcal{P}$	set of equijoin predicates of a query Q_i
$QP = \{QP_1, QP_2, ..., QP_n\}$	set of query plans
sel	join selectivity $\in$ [0..1]
$m(S_i)$	function that returns the memory assigned to S_i tuples

For ease of analysis, we make simplifying assumptions : the tuples are uniformly distributed across the DHT network ; the memory assigned to store tuples is the same at each peer ; we use the average rate to characterize the rate of arrivals of incoming tuples and stream tuples arrive in monotonically increasing order of their timestamps. We use the notations specified in Table 4.1. In order to illustrate our analysis, let us consider the following join query over two streams S_1 and S_2 :

Q : Select *
from S_1[range 5 min], S_2[range 5 min]
where $S_1.x = S_2.x$

The expected tuple arrival rate of streams S_1 and S_2 at each node of the DHT is $\frac{\lambda_1}{n}$ and $\frac{\lambda_2}{n}$ respectively. Thus, the expected number of join tuples generated by S_1 and S_2 over sliding windows at each node can be estimated as

$$T(S_1, S_2) = sel \times (\frac{W_1\lambda_1}{n}) \times (\frac{W_2\lambda_2}{n}) \tag{4.1}$$

Each node needs a memory space for storing tuples in its local sliding window equivalent to $\frac{W_1\lambda_1}{n}$ and $\frac{W_2\lambda_2}{n}$. In general, if $(\frac{W_i\lambda_i}{n} > m(S_i))$ we have a loss rate (Lr) to store tuples equivalent to :

$$Lr(S_i) = \begin{cases} 0, & \frac{W_i\lambda_i}{n} \leq m(S_i) \\ \frac{W_i\lambda_i}{n} - m(S_i), & otherwise \end{cases} \tag{4.2}$$

Assuming that memory is insufficient to retain all the tuples in W_1 and W_2, the loss of join tuples L of S_1 and S_2 is :

$$L(S_1) = sel \times Lr(S_1) \times (\frac{W_2\lambda_2}{n}) \tag{4.3}$$

$$L(S_2) = sel \times Lr(S_2) \times (\frac{W_1\lambda_1}{n}) \tag{4.4}$$

Let α_i be the S_i-tuples stored in the memory space $m(S_i)$ and β_i be the S_i-tuples not stored due to memory constraints (see Figure 4.7). We can rewrite equations (4.3) and (4.4) as :

$$L(S_1) = sel \times \beta_1 \times (\alpha_2 + \beta_2) = (sel \times \alpha_2 \times \beta_1) + (sel \times \beta_1 \times \beta_2)$$
$$L(S_2) = sel \times \beta_2 \times (\alpha_1 + \beta_1) = (sel \times \alpha_1 \times \beta_2) + (sel \times \beta_1 \times \beta_2)$$

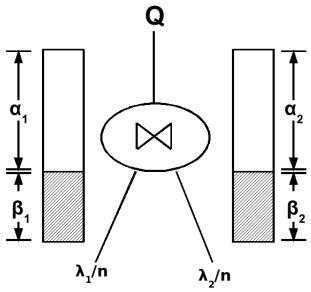


Figure 4.7 – A join state including stored and non stored tuples

Notice that the tuples related to expression $(sel \times \beta_1 \times \beta_2)$ are counted in both $L(S_1)$ and $L(S_2)$. This expression can be rewritten as : $(sel \times Lr(S_1) \times Lr(S_2))$. The total loss of join tuples TL of $S_1 \bowtie S_2$ is the sum of the loss of join tuples $L(S_1)$ and $L(S_2)$ minus the tuples counted twice :

$$TL(S_1, S_2) = L(S_1) + L(S_2) - (sel \times Lr(S_1) \times Lr(S_2)) \tag{4.5}$$

The completeness C of a $S_1 \bowtie S_2$ join query is the fraction of total results $T(S_1, S_2)$ minus the loss of tuples $TL(S_1, S_2)$ and total results $T(S_1, S_2)$, using equation (4.1) and equation (4.5) C is :

$$C = \frac{T(S_1, S_2) - TL(S_1, S_2)}{T(S_1, S_2)} \tag{4.6}$$

Developing expressions in (4.6) allows us to simplify C to :

$$C = \frac{n^2 \times m(S_1) \times m(S_2)}{W_1\lambda_1 \times W_2\lambda_2} \tag{4.7}$$

Moreover, we can write (4.7) as :

$$n = \sqrt{\frac{C \times (W_1\lambda_1) \times (W_2\lambda_2)}{m(S_1) \times m(S_2)}} \tag{4.8}$$

This equation allow us to evaluate how many peers are necessary to process a 2-way join query.

Now we generalize our analysis to m-way joins as following. Recall that the total loss of join tuples TL is the sum of the loss of join tuples minus the tuples counted more than one time. The sum of the loss of join tuples can be easily extended to an m-way join as $\sum_{i=1}^{m} L(S_i)$. However, the expression that represents the tuples counted more than one time is more difficult to generalize. We use the same method of rewriting (4.3) and (4.4) to find the expression that represents the case of tuples counted more than one time. Thus in a $S_1 \bowtie S_2 \bowtie S_3$ join we rewrite $L(S_1)$,$L(S_2)$ and $L(S_3)$, discovering that $(sel^2 \times \beta_1 \times \beta_2 \times \alpha_3)$, $(sel^2 \times \beta_1 \times \beta_3 \times \alpha_2)$ and $(sel^2 \times \beta_2 \times \beta_3 \times \alpha_1)$ are counted twice and $(sel^2 \times \beta_1 \times \beta_2 \times \beta_3)$ is counted triple. Rewriting α_i and β_i we arrive at the following expression : $sel^2Lr(S_1)Lr(S_2)m(S_3) + sel^2Lr(S_1)Lr(S_3)m(S_2) + sel^2Lr(S_2)Lr(S_3)m(S_1) + 2sel^2Lr(S_1)Lr(S_2)Lr(S_3)$.

Repeating the same method with m-way joins ($m \geq 4$) and analyzing the resulting expressions, we arrive at the following general expression for a $S_1 \bowtie S_2 \bowtie .... \bowtie S_m$ join :

$$\sum_{k=2}^{m} \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b))$$

Now, the general case of (4.5) can be expressed as :

$$TL(S_1, S_2, ..., S_m) = \sum_{i=1}^{m} L(S_i) - \sum_{k=2}^{m} \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b)) \tag{4.9}$$

The completeness C of a $S_1 \bowtie S_2 \bowtie .... \bowtie S_m$ join query, using the general form of (4.1) and equation (4.9) is :

$$C = \frac{T(S_1, S_2, ..., S_m) - TL(S_1, S_2, ..., S_m)}{T(S_1, S_2, ..., S_m)} \tag{4.10}$$

Developing expressions in (4.10) allows us to simplify C to :

$$C = \frac{n^m \prod_{i=1}^{m} m(S_i)}{\prod_{i=1}^{m} W_i\lambda_i} \tag{4.11}$$

and to obtain

$$n = \sqrt[m]{\frac{C \times \prod_{i=1}^{m} W_i \lambda_i}{\prod_{i=1}^{m} m(S_i)}} \tag{4.12}$$

It is clear from our analysis that (4.11) is independent of selectivity which is reasonable in the context of continuous join queries. As our analysis shows, DHTJoin can scale up the processing of continuous join queries using multiple peers and improve the completeness of join results. Using (4.12) a DHTJoin user can adjust the size of the network by evaluating how many peers are necessary to process a continuous join query for given stream arrival rates and a desired result completeness.

4.6 Dealing with Data Skew

DHTJoin relies on a hash function to distribute data streams in the DHT using join attribute values. So far, we assumed that the hash function yields uniform distribution of the join attribute values. However, it is well known that parallel join algorithms suffer from join attribute skew, i.e. certain join attribute values are much more frequent than others [ÖV99], which hurts load balancing and thus response time. It is important to note that data skew occurs naturally in many data streaming applications [XKZC08]. For example, in online analysis of transaction logs generated by telephone call records, some numbers used for online tv contests register a huge number of phone calls. In network monitoring applications, malicious traffic traces show that an abnormally high number of source addresses are connected to a single destination address. In DHTJoin, data skew in join attribute values may hurt load balancing of join execution. Furthermore, it may reduce the completeness of join results because the overloaded nodes cannot maintain all received tuples in their sliding window.

In the context of parallel join algorithms, specific solutions have been proposed to deal with data skew. A common solution is to capture join attribute distribution [HL91][WYTD90][KO90]. However, this requires scanning the joined relations before join execution which is not feasible with continuous data streams. The sampling solution proposed in [DNSS92] is also not possible since the arrival of tuples in an arbitrary interleaved way makes the sampling imprecise.

Therefore, we propose a new solution to deal with data skew. The key idea is to distribute the tuples of an overloaded node to some underloaded (or lightly loaded) nodes, called *partners*. There are several issues to address : 1) How to determine that a node is overloaded ; 2) How to find partner(s) node(s) ; 3) What data to migrate and how to execute a join query Q ; and 4) When to start data redistribution.

We say that a node is overloaded if it is not capable of storing all arriving tuples that are not expired. Recall that a load shedding process eliminates stored tuples before they are expired. Each node has a memory space assigned to store tuples belonging to each stream. Thus, detecting whether a node is overloaded is made locally. A node considers that the redistribution of tuples to a *partner* node must begin when a certain threshold δ is exceeded. Let c_n be the storing capacity of node n, i.e. the number of tuples it can store, and s_n be the number of tuples it actually stores. Then, a node n is considered as overloaded when $\frac{s_n}{c_n} > \delta$, where $0 < \delta \leq 1$.

Once a node n becomes overloaded, it should find a *partner* node. For this, it contacts the nodes in its finger table. Each contacted node sends its free storing capacity and the *partner* is the closest node (with smaller latency) whose free capacity is higher than the requirement of n.

If there is no *partner* with enough free capacity to store the tuples of n, several *partners* are chosen from the finger table.

We use the concept of domain partitioning over the join attribute in order to determine what data to send to the *partner(s)* node(s). Consider an attribute a be a join attribute belonging to stream S_i and let D_a be its domain of values. D_a is partitioned into m nonempty sub-domains $d_1, d_2, ..., d_m$ such that their union is equal to D_a and the intersection of any two different sub-domains is empty. When a node is not overloaded, it is responsible for the entire domain D_a of the join attribute. Once a node n is overloaded, the domain is partitioned uniformly into two sub-domains, e.g. d_1 and d_2, and a *partner* node is selected, e.g. n_1. Node n gets responsible for the sub-domain d_1 and *partner* node n_1 gets responsible for the sub-domain d_2. Each overloaded node constructs a local index that stores the upper and lower bounds of the generated sub-domains and the address of their respective responsibles. When one *partner* is not capable of storing the tuples of the overloaded node n, several partners are chosen and the index of n is set accordingly.

If a *partner* n_1 becomes overloaded, it informs n. Then, node n reorganizes the sub-domain of which n_1 is responsible by dividing it and searches among its neighbors for a new responsible n_2. The index is then updated with the new reorganization of sub-domains and node n_1 is contacted in order to inform it that tuples must be sent to node n_2. Thus, the responsability of the sub-domain is shared between n_1 and n_2.

To execute a continuous join query $R \bowtie S$, the overloaded node n executes the same steps as in the non-overloaded case for each incoming tuple $r_i \in R$, with only one additional access to the local index : 1) r_i is used to purge tuples in S stored at the partner node registered in the index, 2) r_i is probed with tuples in S stored at *partner node(s)* registered in the index, and 3) the value of the join attribute of r_i is examinated and r_i is stored in the node indicated by the index. The same steps are executed for a S tuple.

In summary, to obtain a join result DHTJoin must first index each incoming tuple which incurs $O(logN)$ messages (see Section 4.3.2). Then, if a node is overloaded, it redistributes tuples to a *partner* node. The redistribution adds only one message per tuple (to send from the overloaded node to the partner node). Thus, to obtain a join result DHTJoin uses $O(logN) + 1$ messages, keeping the response time slow.

CHAPTER 5

Performance Evaluation

In this section, we provide a performance evaluation of our method through simulation.

Simulator. To test our DHTJoin method, we built a Java-based simulator. Given that DHTJoin relies on a DHT structured overlay network, we choose Chord which is a simple and efficient DHT ; we simulate its routing and churn stabilization protocols. We use a discrete event simulation package SimJava to simulate the distributed processing. To simulate a node, we use a Java object that performs all tasks that must be done by a node wrt the architecure described in Chapter 3. In order to assess our approach, we compare the performance of DHTJoin against a complete implementation of RJoin [ILK08] which is the most relevant related work (see Section 2.2.3). RJoin uses incremental evaluation based on tuple indexing and query rewriting over distributed hash tables. In RJoin a new tuple is indexed twice for each attribute it has ; wrt the attribute name and wrt the attribute value. A query is indexed waiting for matching tuples. Each arriving tuple that is a match causes the query to be rewritten and reindexed at a different node.

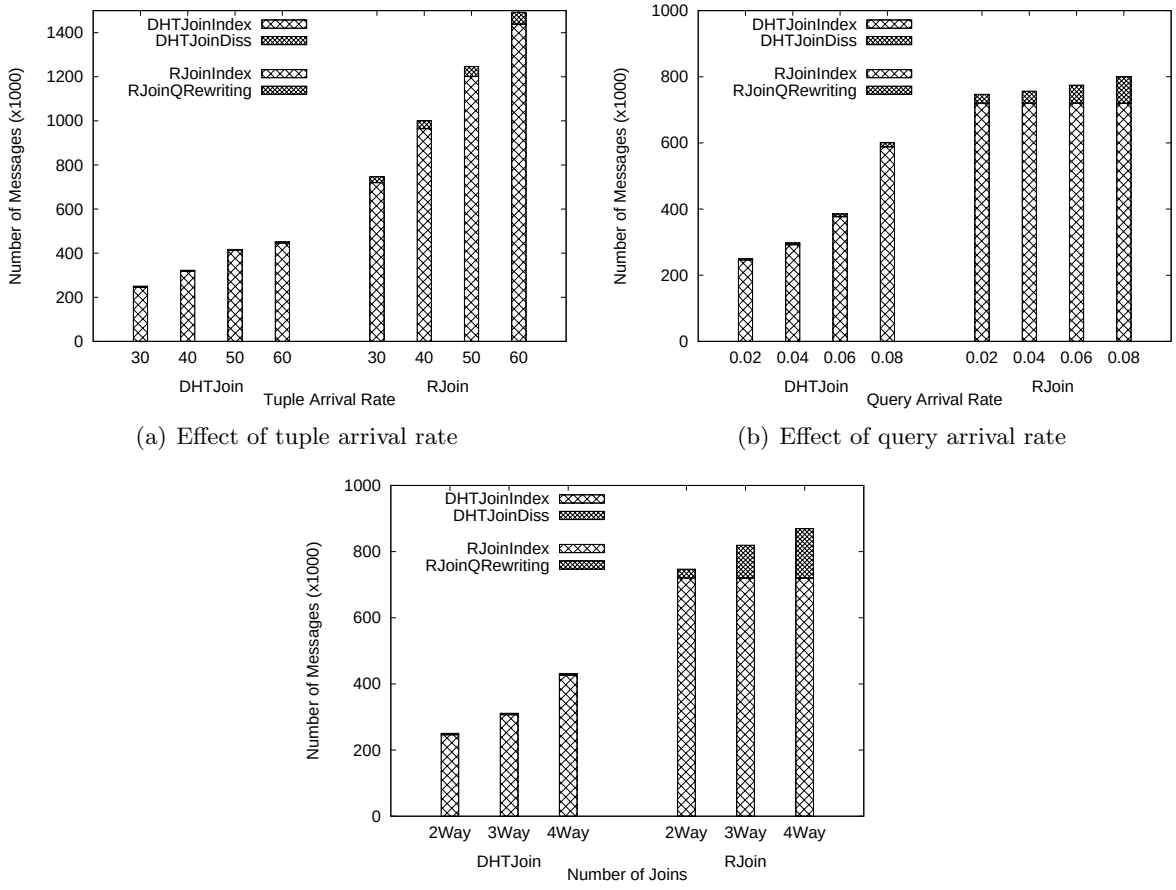
Table 5.1 – Simulation Parameters

Parameter	**Value**
Nb of nodes	1024
Query rate	2-5 queries per 5 minutes
Window size	50 seconds
Nb of streams	2-10
Nb of attributes per stream	2-10
Max value of join attribute	1000
Tuple arrival rate	30-60 tuples per second
Node depart rate	2-5 nodes per 5 minutes
Gossip period T_{gossip}	Adaptive

Data generation. We generate arbitrary input data streams consisting of synthetic asynchronous data items with no tuple-level semantics. We have a schema of 10 relations, each one with 10 attributes. In order to create a new tuple we choose a relation using an uniform distribution and assign values to all its attributes using a Zipf distribution with a default parameter of 0.9. The max value of the domain of the join attribute is fixed to 1000. Unless otherwised specified, tuples on streams are generated at a constant rate varying of $\lambda_i = 30$ tuples/second to $\lambda_i = 60$ tuples/second.

Query generation. We generate queries of type 1 and 2 as specified in Section 4.1. Unless otherwise specified, the window size is set to 50 seconds. We generate a default probing sequence for relations in queries of type 1 and bushy plans are generated for queries of type 2. This information is used by the nodes in order to set local join states. Probing sequences can be adapted locally during the execution of queries.

In the rest of this section, we evaluate network traffic and the effectiveness of the approaches proposed in Section 4.4 and in Section 4.6 to deal with node failures and skewed data respectively. The main simulation parameters are summarized in Table 5.1.



(a) Effect of tuple arrival rate

(b) Effect of query arrival rate

(c) Effect of number of joins

Figure 5.1 – Effect of tuple, query arrival rates and number of joins on the network traffic

5.1 Network Traffic

In this section, we investigate the effect of tuples' arrival rate, query's arrival rate and number of joins on the network traffic. The network traffic of RJoin and DHTJoin grows as the tuples' arrival rate grows. In RJoin, as more tuples arrive, the number of messages related

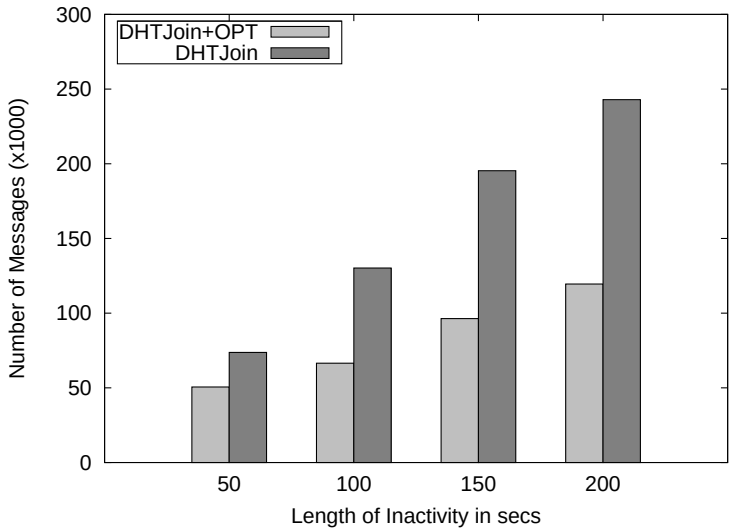


Figure 5.2 – Reduction of intermediate results and its impact on network traffic

to the indexing of tuples and query rewriting increases (see Figure 5.1(a)). DHTJoin generates significantly less messages than RJoin. The reason is that before indexing a tuple, DHTJoin checks for the existence of a query that requires it, but RJoin indexes all tuples twice (even if there is no query for them). In Figure 5.1(b), we show that, as more queries arrive, RJoin generates more query rewriting messages. However, DHTJoin generates more messages only if new queries related to new different values used in the tuples arrive in the system. Figure 5.1(c) shows that more join require more network traffic. RJoin generates more query rewriting when there are more joins in the queries. However, in DHTJoin the network traffic increases only if the arriving queries require attributes that are not present in the already disseminated queries. The reason is that with the dissemination of queries, DHTJoin can avoid the unnecessary indexing of tuples that are not required by the queries.

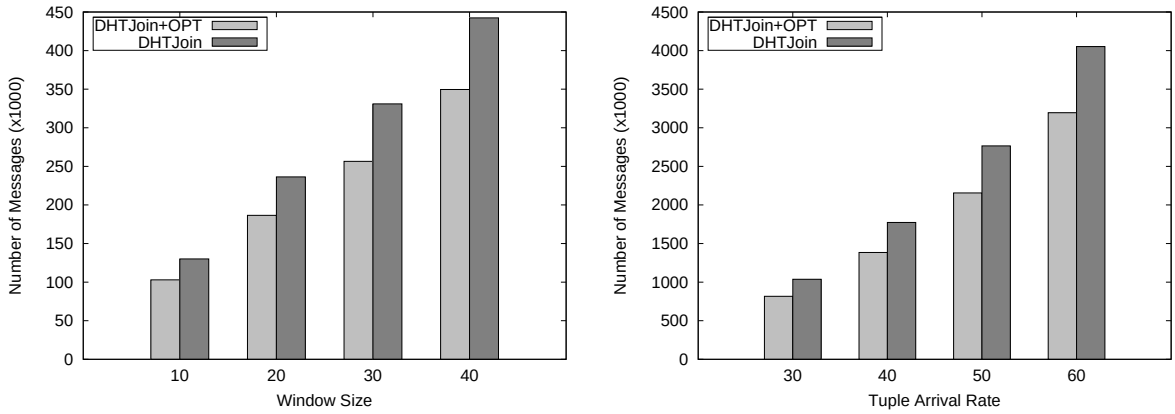
In summary, due to the integration of query dissemination and hash-based placement of tuples our approach avoids the excessive traffic generated by RJoin which is due to its method of indexing tuples.

5.2 Node failures during query execution

We now investigate the effect of the approach proposed in Section 4.4.2 in order to deal with node failures. Query execution with this approach is referred as DHTJoin+OPT. In our experiment, tuples arrive at $\lambda_i = 60$ tuples/sec and a query of type 2 with query plan $(A \bowtie B) \bowtie (C \bowtie D)$ is executed. In Figure 5.2, we show that, as the period of inactivity (time between fail and recovery) of a stream source gets longer, the generation of tuples that never contribute to join results increases. However, by eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic.

Figure 5.3(a) shows the number of messages generated while varying the window size of the query. During the execution of the query, the 20% of the nodes producing $C \bowtie D$ tuples fail. As

we can see, a longer window leads to a larger number of total intermediate results since a tuple has more chance of participate in a join match. However, with DHTJoin+OPT a reduction of up to 23% of network traffic is obtained.



(a) Effect of window size

(b) Effect of tuple arrival rate

Figure 5.3 – Effect of window size and tuple arrival rate on the network traffic

Figure 5.3(b) shows the number of messages while increasing the tuple rate. Intuitively, a rapid tuple rate leads to more intermediate results, many of which are not demanded by their corresponding consumers when nodes fail. As is shown in Figure 5.3(b), the effect of the tuple rate is similar to that of window size. Thus, a reduction of unnecessary intermediate results is obtained.

5.3 Node failures during query dissemination

The failure of a node in the tree structure generated by the dissemination procedure makes the entire subtree under this node unreachable. To provide reliability in the dissemination of queries, we proposed a gossip based protocol (see Section 4.4.1).

To evaluate the effectiveness of our approach regarding an increment of node's failure rate we originate queries every 100 seconds on average and we increment the node's failure rate (see Figure 5.4) during 1 hour. We consider a first scenario where the queries are disseminated using the technique described in Section 4.3.1 and a second scenario where the queries are disseminated using the same technique in complement with gossip. Figure 5.4 shows that with node failures the dissemination cannot achieve a network coverage of 100%. However, the dissemination of queries complemented with gossip can obtain a network coverage of 100% in spite of an increase in node failures.

The first experiment evaluates the effectiveness of a gossip protocol in order to obtain a network coverage of 100% when nodes fail. In the next experiment, we are interested in the impact (overhead) of the gossip protocol in the network traffic wrt a dissemination without node failures. We vary the gossip period (T_{gossip}) ; then after 1 simulation hour, we collect the number

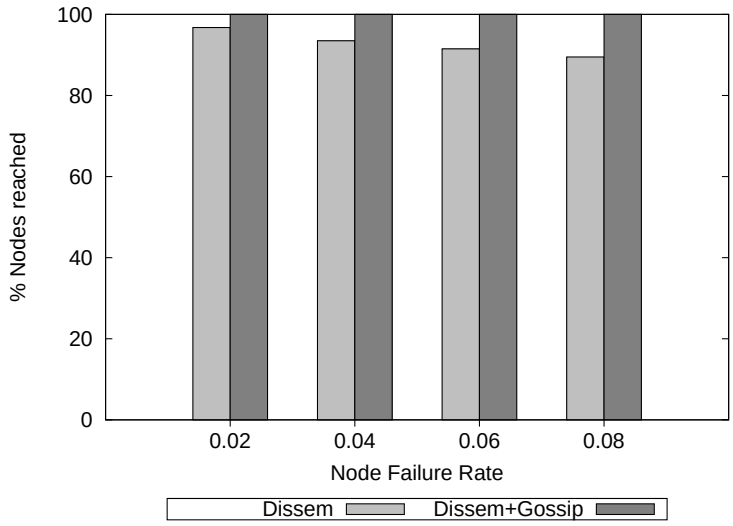


Figure 5.4 – Effect of dealing with node failures during the dissemination of queries

of messages produced by the gossip protocol and calculate the overhead. When decreasing the T_{gossip} parameter, gossip exchanges are less spaced and thus more frequent. Thus, as shown in Table 5.2, the gossip protocol generates more messages and its overhead increases. However, increasing T_{gossip} can reduce the effectiveness of the gossip protocol. Therefore, we evaluate the approach proposed in Section 4.4.1 where T_{gossip} is particularly adapted at each node. The results obtained confirm the effectiveness of our appraoch showing that adapting T_{gossip} reduces the overheah of the gossip protocol.

Table 5.2 – Impact of Gossip using static and adjustable T_{gossip}.

T_{gossip}	#Messages	Overhead
Adaptive	1992	16.2%
300_{sec}	5103	40.86%
250_{sec}	6245	49.55%
200_{sec}	9459	74.22%

The impact of the query's arrival rate is shown in Figure 5.5. We vary this parameter from 2 to 5 queries each 5 minutes and set the node's depart rate to 2 nodes each 5 minutes. As we can see, more queries produce more dissemination messages. However, as is shown in Figure 5.5 the number of messages produced by the gossip protocol is more or less constant. Thus, the overhead produced by the gossip protocol decreases while the query's arrival rate increases (see Table 5.3). This shows that gossip exchanges contains much more relevant information when more queries arrive allowing nodes to obtain information about queries not received due to node failures during the dissemination.

The impact of the node failure's rate is shown in Figure 5.6. During 1 simulation hour

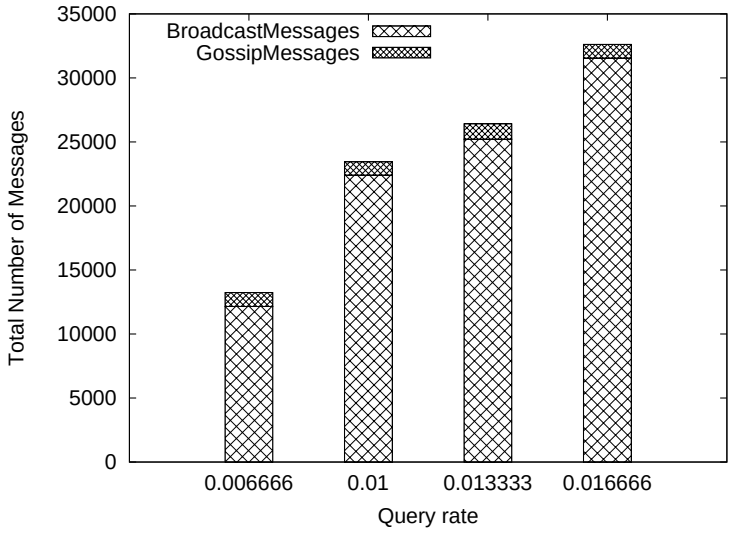


Figure 5.5 – Effect of query arrival rate

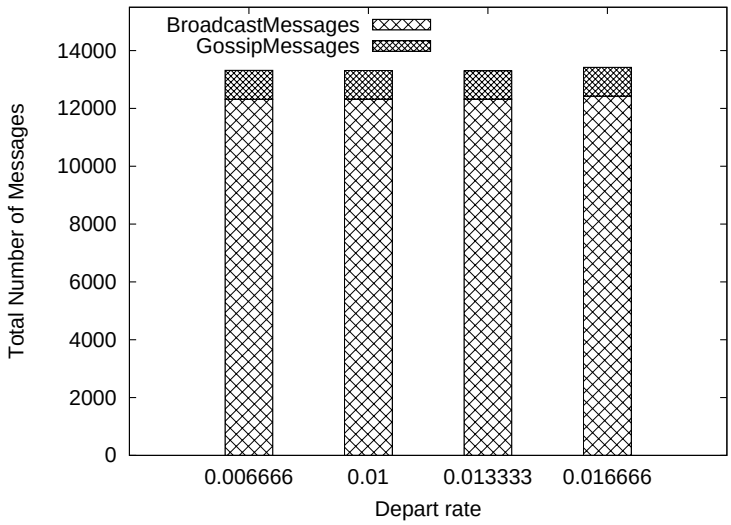


Figure 5.6 – Effect of node failure rate

this parameter is varied from 2 to 5 failed nodes each 5 minutes generating a very dynamic environment. As is shown in Figure 5.6, the number of messages due to gossip exchanges and dissemination remains constant. However, the overhead increases while the depart of nodes increases (see Table 5.4). This situation is originated during the dissemination of queries since as more nodes fail more fingers tables are not up to date. Thus, the dissemination algorithm can not assure a dissemination free of redundant messages (see Table 5.4). Since the problem is caused by the content of the finger table, the stabilization protocol of Chord must be tuned (what is out of the scope of this work). The gossip protocol can not avoid these messages, thus it suggests a more frequent execution of the stabilization protocol incrementing the number of messages.

Table 5.3 – Overhead due to gossip messages

Queries/5*min*	#Gossip messages	Overhead
2	1082	7.79%
3	1054	4.20%
4	1207	3.29%
5	1078	2.85%

Table 5.4 – Effect of node depart rate

Departs/5*min*	#Broadcast msgs	#Broadcast redundant msgs	#Gossip msgs	Overhead
2	12316	184	1003	8.49%
3	12318	249	995	10.47%
4	12317	302	989	10.85%
5	12422	392	1000	11.52%

However, concerning the gossip protocol and its strategy to provides an adaptive T_{gossip} we shown that it provides an efficient solution to the problem of node failures during the dissemination of queries.

5.4 Data Skew

The distribution of tuples on their joining attribute is vulnerable to the presence of skew in the underlying data hurting load balancing of join execution and reducing completeness of join results. To provide a solution to this problem we propose an approach based on *partners* nodes that store the tuples of an overloaded node (see Section 4.6).

To generate test data we use the ideas proposed in [DNSS92] where an alternative method to drawn attribute values from a Zipf distribution is prefered. The argument is that the data must be generated in order to make the experiments much easier to understand and control. The idea is to keep the same behaviour of the Zipf distribution (a small number of highly skewed values with the bulk of the values appearing very infrequently). Thus, we generate tuples as follows. We assume that the stream S_i has an arrival rate of λ_i and we consider a query with a sliding window of size W. We use the equation (4.11) in order to set the DHT and calculate c_n (the storing

capacity of a node). We generate skewed attribute values during a time interval equivalent to $f \times \frac{c_n}{\lambda_i}$, where $f = \{1, 1.1, 1.2, 1.4\}$. Since we known the identifiers of the DHT nodes, we can generate skewed values during a specified time interval in order to overload certains nodes. Thus, we configure various patterns of overload and during the same interval of time certains nodes get overloaded.

Table 5.5 – Average time to obtain a result tuple

Query type	f=1	f=1.1	f=1.2	f=1.4
type 1	0.44 sec	0.42 sec	0.46 sec	0.47 sec
type 2	0.45 sec	0.47 sec	0.43 sec	0.48 sec

In our experiments, we consider a DHT of 16 nodes and a query of type 1 $A \bowtie B$ with a sliding windows of size $W = 5$ minutes. Tuples of streams A and B arrive at $\lambda_i = 30$ tuples/sec. Thus, using the equation (4.11) the value of c_n is 563 tuples. We consider that a node is overloaded if $\frac{s_n}{c_n} > 0.9$ and the required completeness is 100%, i.e. $C = 1$. We consider that all streams present skewed values and the join attribute takes a value between 0 and 128.

We measure the average time to obtain a join result in a *partner* node. We set the latency of a direct communication between any two nodes to 100 ms. We run the experiments 10 times changing the stream sources. Recall that a lookup operation in Chord takes $O(log\ n)$ on average. Since the latency of each hop is 100 ms., the average lookup latency is $O(log\ n) \times 100 = 0.4$ secs when $n = 16$. Considering the additional message in order to find the node that stores the tuples of the overload node, analytically the average time to obtain a tuple is $0.4 + 0.1 = 0.5$ secs.

Table 5.5 shows the average time obtained by the simulator to generate a result tuple from a *partner* node in a DHT with $n = 16$ nodes. We see that the obtained values from the simulation are very close to the analytical average time under all overload patterns.

In the second experiment we evaluate the efectiveness of our approach using a query of type 2 with query plan $(A \bowtie B) \bowtie (C \bowtie D)$. In the case of a query type 2 we generate skewed data for all the streams of the query. The average times obtained are very close to the values reported for the queries of type 1 because the join operators $(A \bowtie B)$ and $(C \bowtie D)$ are executed en parallel.

In summary, with $\delta = 0.9$ we prevent the lost of tuples due to a lack of space to store them. Thus, result completeness is not compromised and the approach to deal with skew data based on *partners* nodes keeps the response time low.

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

P2P networks is an important architecture to support distributed stream applications. Complex query operators such as a join operator are essential for various applications such as network monitoring, electronic trading markets, online analysis of telephone call records and many others. Although the execution of join queries has been widely studied in the literature, the unbounded nature of data streams and the distributed processing of join queries under a P2P environment creates new challenges.

In this thesis, we proposed a new method, called DHTJoin, for processing continuous join queries using structured P2P networks in DSMS. DHTJoin combines hash-based placement of tuples and dissemination of queries using the trees formed by the underlying DHT links. DHTJoin takes advantage of the indexing power of DHT protocols and dissemination of queries to avoid the placement of tuples that cannot contribute to generate join results. DHTJoin deals with node failures and problems generated by skewed data.

We have presented (Chapter 3) the architecture of DHTJoin, which has been designed to meet the challenges of efficiently executing continuous queries in an structured P2P environment. This architecture is deployed on top of a DHT protocol which influences tasks such as query dissemination and flow partitioning. The dissemination of queries is performed constructing a tree rooted at the node submitting the query and using the information stored in the routing table of nodes. The design of DHTJoin is based on Chord but we show in Section 4.3.4 that the dissemination technique can be adapted to others DHTs such as Pastry [RD01] and Tapestry [ZHS+04]. The tuples are partitioned by value where the DHT is used for routing tuples and as a hash table for storing tuples. However, once tuples arrive at the system they are filtered. This filtering process drops tuples not concerned by queries as early as possible achieving significant performance gains in terms of network traffic. This is more efficient than the approaches based on structured P2P overlays, e.g. PIER [HHL+03] and RJoin [ILK08], which typically index all tuples in the network.

The DHTJoin method is presented in Chapter 4, specifying the type of join queries processed by DHTJoin and the approaches for parallelization of queries. For queries of type 1 the partitioned parallelism gives more scalability and its combination with the MJoin operator gives more flexibility face to variations in the workload because the shape of the query plan residing on individual machines is restructured easy and independently. On the other hand, queries of type 2 are parallelized using pipelined parallelism using segmented bushy trees. This allow the execution of queries applying independent parallelism with minimal dependencies among subtrees. This strategy offers a good trade-off between partitioned and pipelined parallelism for complex

multi-join queries while giving more opportunities to share executions plans with queries of type 1.

We tackle the problem of node failures during the dissemination of queries using a gossip protocol based on the concept of anti-entropy. This gossip protocol allows the detection and correction of inconsistencies about the knowledge of queries in the system by continuous gossiping. Basically, the gossip algorithm is deployed on top of the DHT using the information stored in the *Finger* table of nodes for random gossip exchange. This level of determinism on the choice of which nodes to gossip reduce redundant messages while achieving complete dissemination under node failures. Node failures during query execution is tackled through the collaboration between consumer and producer operators eliminating unnecessary intermediate tuples that do not contribute to join results. Thus, a significant savings in terms of network traffic is obtained.

DHTJoin provides an efficient solution to deal with overloaded nodes as a result of data skew. An overload node use the information stored in its *Finger* table to choose some underload node to which tuples of the overloaded node are distributed. We show that, in this case, DHTJoin incurs only one additional message per joined tuple produced, thus keeping response time low.

We propose a theoretical approach which relate peer memory constraints, stream arrival rates, and result completeness. We state that could be useful to a DHTJoin user (e.g. an application developer) to define and tune a DHT network for specific application requirements. We showed analytically that DHTJoin can scale up the processing of continuous join queries using multiple peers and improves the completeness of join results.

To validate our contribution, we implemented DHTJoin as well as RJoin which is the most relevant state of the art solution in the context of processing continuous join queries using DHTs. Our performance evaluation shows that DHTJoin yields significant performance gains due to the mechanism of indexing tuples and the elimination of unnecessary intermediate results. Our results also demonstrate that the total number of messages of DHTJoin is always less than that of RJoin wrt tuple arrival rate, query arrival rate and number of joins. We also show that the problem of node failures during the dissemination of queries can be complemented with a gossip based protocol that allows, in spite of node failures, a network coverage of 100%.

6.2 Future Work

Our future work focuses on adding to DHTJoin more advances features in terms of query processing. For example, Skyline queries [BKS01][JEHH07] are important to applications such as multi-criteria decision making where query results are obtained with respect to user preferences. Due to its importance, research efforts have been started to implement a skyline operator in commercial DBMS [CDK06]. In a d-dimensional data space, a skyline query retrieves those data points that are not *dominated* by other points. A point dominates another point if it as good or better in all dimensions and better in at least one dimension. More formally, a point $p = (p[1], p[2], ..., p[d])$ where $p[i]$ is a value on dimension d_i dominates another point $q = (q[1], q[2], ..., q[d])$ iff $p[i] \leq q[i]$ $1 \leq i \leq d$ and there is at least one dimension j such that $p[j] < q[j]$

Figure 6.1 shows an example of skyline over a small set of points $\{p_1, p_2, p_3, p_4, p_5\}$. We can see that p_2 dominates p_1 since both coordinates of p_2 are smaller than that of p_1. Thus, p_1 cannot be a skyline point. Since p_2, p_3 and p_4 are not dominated by any other points, they are the skyline in this set of points.

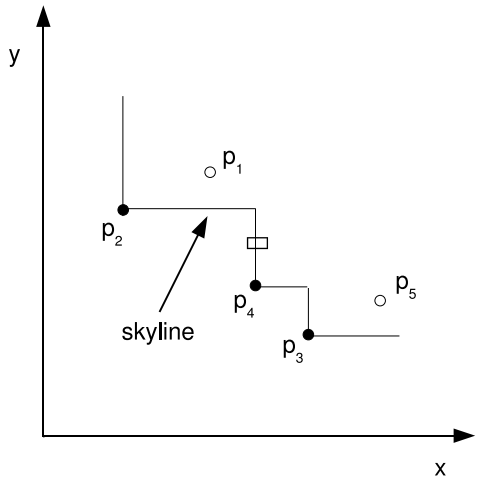


Figure 6.1 – A skyline example

Existing research assumes that the skyline query is processed using attributes from the same table. However, there are scenarios where the attributes that users are interested in are from different tables. In this case a skyline query access data from distributed sources via joins and combines several attributes across these sources through user preferences. State of the art algorithms that operate using indices constructed on the skyline attributes [PTFS03] are inappropriate in this setting, since the construction of an index on the fly can be expensive.

Considering a distributed setting, skyline queries processing in super-peers networks has been proposed in [VDKV07] and the approach proposed in [WOTX07] extends BATON [JOV05] to support skylines queries. However, these approaches not consider joins.

To the best of our knowledge, only a few research works tackle skyline queries in the presence of joins [SWLT08][JMP+10]. However, these approaches not consider continuous queries and obtain skyline point over static queries and data.

Considering the state of the art w.r.t. skylines queries, we plan as a future work to address the problem of processing distributed continuous skyline join queries. Skyline operator is by definition a blocking operator and its integration with join in a distributed environment is a challenge.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. Stream : The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1) :19–26, 2003.

[ABC+03] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, and Tova Milo. Dynamic xml documents with distribution and replication. In *SIGMOD Conference*, pages 527–538, 2003.

[ACc+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora : a new model and architecture for data stream management. *VLDB J.*, 12(2) :120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road : A stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@. *Commun. ACM*, 45(11) :56–61, 2002.

[ACMD+03] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid : a self-organizing structured p2p system. *SIGMOD Record*, 32(3) :29–33, 2003.

[ADGK07] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Vana Kalogeraki. Efficient approximate query processing in peer-to-peer networks. *IEEE Trans. Knowl. Data Eng.*, 19(7) :919–933, 2007.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies : Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[APV06] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Reducing network traffic in unstructured p2p systems using top- queries. *Distributed and Parallel Databases*, 19(2-3) :67–86, 2006.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4) :335–371, 2004.

[AW04] Arvind Arasu and Jennifer Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3) :6–12, 2004.

[BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury : supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[BBMS08] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008.

[BGGMM04] Mayank Bawa, Aristides Gionis, Hector Garcia-Molina, and Rajeev Motwani. The price of validity in dynamic networks. In *SIGMOD Conference*, pages 515–526, 2004.

[BGMGM03] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab, April 2003.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[BHÖ+99] Kenneth P. Birman, Mark Hayden, Öznur Özkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2) :41–88, 1999.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13 :422–426, 1970.

[BMM+04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.

[BMWM05] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[BT03] Peter A. Boncz and Caspar Treijtel. Ambientdb : Relational query processing in a p2p network. In *DBISP2P*, pages 153–168, 2003.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq : Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[CDK06] Surajit Chaudhuri, Nilesh N. Dalvi, and Raghav Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, page 64, 2006.

[CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock : a language for extracting signatures from data streams. In *KDD*, pages 9–17, 2000.

[CG07] Graham Cormode and Minos N. Garofalakis. Streaming in a connected world : querying and tracking distributed data streams. In *SIGMOD Conference*, pages 1178–1181, 2007.

[CJK+03] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony I. T. Rowstron, Marvin Theimer, Helen J. Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM*, 2003.

[CJSS03] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope : A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB*, pages 25–30, 2004.

[Des04] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1) :44–49, 2004.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[DHJ+05] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In *Peer-to-Peer Computing*, pages 57–66, 2005.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[DZD+03] Frank Dabek, Ben Y. Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *IPTPS*, pages 33–44, 2003.

[EAABH03] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured p2p networks. In *IPTPS*, pages 304–314, 2003.

[FKL+09] Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Étienne Rivière, and Spyros Voulgaris. Pulp : Un protocole épidemique hybride. In *AlgoTel*, 2009.

[GAA03] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[GGG+03] P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In *SIGCOMM*, pages 381–394, 2003.

[GJK+08] Lukasz Golab, Theodore Johnson, Nick Koudas, Divesh Srivastava, and David Toman. Optimizing away joins on data streams. In *SSPS*, pages 48–57, 2008.

[GÖ03a] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2) :5–14, 2003.

[GÖ03b] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[Has95] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Departament of Computer Science, Stanford, USA, December 1995.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *VLDB*, pages 321–332, 2003.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet : A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[HL91] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[HRM08] Cecilia Hernández, M. Andrea Rodríguez, and Mauricio Marín. Complex queries for moving object databases in dht-based systems. In *Euro-Par*, pages 424–433, 2008.

[ILK08] Stratos Idreos, Erietta Liarou, and Manolis Koubarakis. Continuous multi-way joins over distributed hash tables. In *EDBT*, pages 594–605, 2008.

[JEHH07] Wen Jin, Martin Ester, Zengjian Hu, and Jiawei Han. The multi-relational skyline operator. In *ICDE*, pages 1276–1280, 2007.

[JMB05] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3) :219–252, 2005.

[JMP+10] Wen Jin, Michael D. Morse, Jignesh M. Patel, Martin Ester, and Zengkian Hu. Evaluating skylines in the presence of equi-joins. In *ICDE*, pages –, 2010.

[JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton : A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[KMR02] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. Sos : secure overlay services. In *SIGCOMM*, pages 61–72, 2002.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[KO90] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash : A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, pages 210–221, 1990.

[KSH+08] Marcel Karnstedt, Kai-Uwe Sattler, Michael Haß, Manfred Hauswirth, Brahmananda Sapkota, and Roman Schmidt. Estimating the number of answers with guarantees for structured queries in p2p databases. In *CIKM*, pages 1407–1408, 2008.

[KvS07] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *Operating Systems Review*, 41(5) :2–7, 2007.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, pages 84–95, 2002.

[LR05] Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840, 2005.

[LS03] Alberto Lerner and Dennis Shasha. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Data Eng. Bull.*, 26(1) :49–56, 2003.

[LSSP03] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@home and genome@home : Using distributed computing to tackle previously intractable problems in computational biology. *Modern Methods in Computational Biology, Horizon Press*, 2003.

[LW06] Xiuqi Li and Jie Wu. Searching techniques in peer-to-peer networks. In S. Shi W. Sheng, J. Wu, editor, *Handbook on Theoretical and Algorithmic Aspects of Ad Hoc, Sensor and Peer-to-Peer Networks*. 2006.

[LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, pages 492–503, 2004.

[MAA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

[MK02] Daniel A. Menascé and Lavanya Kanchanapalli. Probabilistic scalable p2p resource location services. *SIGMETRICS Performance Evaluation Review*, 30(2) :48–58, 2002.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, 1986.

[MM02] P. Maymounkov and D. Mazieres. Kademlia : A Peer-to-Peer Information System Based on the XOR Metric. *Peer-To-Peer Systems : First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002*, 2002.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy : a scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002.

[NFL04] Felix Naumann, Johann Christoph Freytag, and Ulf Leser. Completeness of integrated information sources. *Inf. Syst.*, 29(7) :583–615, 2004.

[NOTZ03] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Peerdb : A p2p-based system for distributed data sharing. In *ICDE*, pages 633–644, 2003.

[NSS03] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design issues and challenges for rdf- and schema-based peer-to-peer systems. *SIGMOD Record*, 32(3) :41–46, 2003.

[NWQ+02a] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella : a p2p networking infrastructure based on rdf. In *WWW*, pages 604–615, 2002.

[NWQ+02b] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella : a p2p networking infrastructure based on rdf. In *WWW*, pages 604–615, 2002.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[PRR99] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3) :241–280, 1999.

[PTFS03] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.

[RBR+04] Mema Roussopoulos, Mary Baker, David S. H. Rosenthal, Thomas J. Giuli, Petros Maniatis, and Jeffrey C. Mogul. 2 p2p or not 2 p2p ? In *IPTPS*, pages 33–43, 2004.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–, 2003.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[RK02] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *INFOCOM*, 2002.

[RRHS04] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief announcement : prefix hash tree. In *PODC*, page 368, 2004.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[SSDN02] Mario T. Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup. Technical report, Stanford University, 2002.

[Sul96] Mark Sullivan. Tribeca : A stream database manager for network traffic analysis. In *VLDB*, page 594, 1996.

[SWLT08] Dalie Sun, Sai Wu, Jianzhong Li, and Anthony K. H. Tung. Skyline-join in distributed databases. In *ICDE Workshops*, pages 176–181, 2008.

[TcZ07] Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik. Staying fit : Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.

[TIM+03] Igor Tatarinov, Zachary G. Ives, Jayant Madhavan, Alon Y. Halevy, Dan Suciu, Nilesh N. Dalvi, Xin Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. *SIGMOD Record*, 32(3) :47–52, 2003.

[TR03a] Dimitrios Tsoumakos and Nick Roussopoulos. A comparison of peer-to-peer search methods. In *WebDB*, pages 61–66, 2003.

[TR03b] Dimitrios Tsoumakos and Nick Roussopoulos. Estimating aggregates on a peer-to-peer network. Technical Report 4451, University of Maryland, April 2003.

[UF00] Tolga Urhan and Michael J. Franklin. Xjoin : A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2) :27–33, 2000.

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2) :218–246, 1987.

[VATS04] Vasileios Vlachos, Stephanos Androutsellis-Theotokis, and Diomidis Spinellis. Security applications of peer-to-peer networks. *Computer Networks*, 45(2) :195–205, 2004.

[VDKV07] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Michalis Vazirgiannis. Skypeer : Efficient subspace skyline computation over distributed data. In *ICDE*, pages 416–425, 2007.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[VP04] Patrick Valduriez and Esther Pacitti. Data management in large-scale p2p systems. In *VECPAR*, pages 104–118, 2004.

[VvS07] Spyros Voulgaris and Maarten van Steen. Hybrid dissemination : Adding determinism to probabilistic multicasting in large-scale p2p systems. In *Middleware*, pages 389–409, 2007.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3) :38–49, 1992.

[WOTX07] Shiyuan Wang, Beng Chin Ooi, Anthony K. H. Tung, and Lizhen Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, pages 1126–1135, 2007.

[WRC00] Marc Waldman, Aviel Rubin, and Lorrie Cranor. Publius : A robust, tamper-evident, censorship-resistant web publishing system. In *In Proc. 9th USENIX Security Symposium*, pages 59–72, 2000.

[WRGB06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice : New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.

[WS92] Michael S. Warren and John K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *SC*, pages 570–576, 1992.

[WYTD90] Joel L. Wolf, Philip S. Yu, John Turek, and Daniel M. Dias. An effective algorithm for parallelizing hash joins in the presence of data skew. Wishful Research Result RC 15510, IBM T.J. Watson Research Center, 1990.

[XKZC08] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD Conference*, pages 1043–1052, 2008.

[XL04] Li Xiong and Ling Liu. Peertrust : Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Knowl. Data Eng.*, 16(7) :843–857, 2004.

[YG03] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[YGM02] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.

[YP08] Yin Yang and Dimitris Papadias. Just-in-time processing of continuous queries. In *ICDE*, pages 1150–1159, 2008.

[ZHC08] Runfang Zhou, Kai Hwang, and Min Cai. Gossip trust for fast reputation aggregation in peer-to-peer networks. *IEEE Trans. Knowl. Data Eng.*, 20(9) :1282–1295, 2008.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry : a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1) :41–53, 2004.

[Zhu06] Y. Zhu. *Dynamic Optimization and Migration of Continuous Queries Over Data Streams*. PhD thesis, Worcester Polytechnic Institute, Worcester, MA, USA, August 2006.

[ZR07] Yali Zhu and Elke A. Rundensteiner. Adapting partitioned continuous query processing in distributed systems. In *ICDE Workshops*, pages 594–603, 2007.

[ZYYZ06] Yongluan Zhou, Ying Yan, Feng Yu, and Aoying Zhou. Pmjoin : Optimizing distributed multi-way stream joins by stream partitioning. In *DASFAA*, pages 325–341, 2006.

Traitement de requêtes de jointures continues dans les systèmes pair-à-pair (P2P) structurés

Wenceslao Enrique Palma Munoz

Résumé

De nombreuses applications distribuées partagent la même nécessité de traiter des flux de données de façon continue, par ex. la surveillance de réseau ou la gestion de réseaux de capteurs. Dans ce contexte, un problème important et difficile concerne le traitement de requêtes continues de jointure qui nécessite de maintenir une fenêtre glissante sur les données la plus grande possible, afin de produire le plus possible de résultats probants. Dans cette thèse, nous proposons une nouvelle méthode pair-à-pair, DHTJoin, qui tire parti d'une Table de Hachage Distribuée (DHT) pour augmenter la taille de la fenêtre glissante en partitionnant les flux sur un grand nombre de noeuds. Contrairement aux solutions concurrentes qui indexent tout les tuples des flux, DHTJoin n'indexe que les tuples requis pour les requêtes et exploite, de façon complémentaire, la dissémination de requêtes. DHTJoin traite aussi le problème de la dynamicité des noeuds, qui peuvent quitter le système ou tomber en panne pendant l'exécution. Notre évaluation de performances montre que DHTJoin apporte une réduction importante du trafic réseau, par rapport aux méthodes concurrentes.

Mots-clés : Systèmes pair-à-pair, Traitement de requêtes

Abstract

Recent years have witnessed the growth of a new class of data-intensive applications that do not fit the DBMS data model and querying paradigm. Instead, the data arrive at high speeds taking the form of an unbounded sequence of values (data streams) and queries run continuously returning new results as new data arrive. In these applications, data streams from external sources flow into a Data Stream Management System (DSMS) where they are processed by different operators. Many applications share the same need for processing data streams in a continuous fashion. For most distributed streaming applications, the centralized processing of continuous queries over distributed data is simply not viable. This research addresses the problem of computing continuous join queries over distributed data streams. We present a new method, called DHTJoin that exploits the power of a Distributed Hash Table (DHT) combining hash-based placement of tuples and dissemination of queries by exploiting the embedded trees in the underlying DHT, thereby incuring little overhead. Unlike state of the art solutions that index all data, DHTJoin identifies, using query predicates, a subset of tuples in order to index the data required by the user's queries, thus reducing network traffic. DHTJoin tackles the dynamic behavior of DHT networks during query execution and dissemination of queries. We provide a performance evaluation of DHTJoin which shows that it can achieve significant performance gains in terms of network traffic.

Keywords: P2P Systems, Continuous Query Processing

acm Classification

Categories and Subject Descriptors : H.2.4 [**Database Management**]: Systems—*Distributed databases, Query processing.*

Discipline : INFORMATIQUE
Specialité : Base de Données

Laboratoire : **L**aboratoire d'**I**nformatique de **N**antes **A**tlantique.
cnrs fre 2729. 2, rue de la Houssinire, bp 92 208 – 44 322 Nantes, Cedex 3.