Incremental Evaluation of Continuous Analytic Queries in a High-Level Query Language

Petros Zervoudakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Incremental Evaluation of Continuous Analytic Queries in a High-Level Query Language

Thesis submitted by
Petros Zervoudakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Petros Zervoudakis

Committee approvals: ______________________
Dimitris Plexousakis
Professor, Thesis Supervisor

____
Nicolas Spyratos
Professor Emeritus, Thesis Co-Supervisor

____
Yannis Tzitzikas
Associate Professor, Committee Member

Departmental approval: ______________________
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, February 2020

Abstract

Data analytics have received a significant attention in recent years, as huge amounts of data is generated each day from various sources. Analysis of these massive data poses an interesting but challenging task and requires new forms of processing to enable enhanced decision making, insight discovery and process optimization. In addition, besides their ever increasing volume, data sets change frequently, and as such, results to continuous queries have to be updated at short intervals. In this thesis, we address the problem of evaluating continuous queries over big data streams that are frequently updated. To this end, we adopt HIFUN, a high-level query language, proposed for expressing analytic queries over big data sets. HIFUN offers a clear separation between the conceptual layer, where analytic queries are defined independently of the nature and location of data, and the physical layer where queries are evaluated, by encoding them as map-reduce jobs or as SQL group-by queries, thus supporting different types of data set formats. Using HIFUN, we design an algorithm for incremental evaluation of continuous queries, processing only the most recent data batch, and exploiting already computed information, without requiring the evaluation of the query over the complete data set. Subsequently, we translate the generic algorithm to both SQL and MapReduce using SPARK, exploiting the query rewriting methods provided by HIFUN. Using a synthetic data set, we demonstrate the effectiveness of our approach in achieving query answering efficiency. Finally, we show that by exploiting the formal query rewriting methods of HIFUN, we can further reduce the computational cost, adding another layer of query optimization in our implementation.

Περίληψη

Η διαδικασία ανάλυσης δεδομένων έχει λάβει σημαντική προσοχή τα τελευταία χρόνια καθώς τεράστιες ποσότητες δεδομένων παράγονται καθημερινά από διάφορες πηγές. Η ανάλυση αυτών των τεράστιων δεδομένων αποτελεί ένα ενδιαφέρον αλλά και δύσκολο έργο και απαιτεί νέες μορφές επεξεργασίας ώστε να είναι εφικτή η λήψη αποφάσεων, η ανακάλυψη γνώσεων και η βελτίωση των διαδικασιών. Επιπλέον, εκτός από τον συνεχώς αυξανόμενο όγκο τους, τα σύνολα δεδομένων αλλάζουν συνεχώς, και ως εκ τούτου, τα αποτελέσματα σε συνεχόμενα ερωτήματα πρέπει να ενημερώνονται σε σύντομα χρονικά διαστήματα. Σε αυτή την εργασία, αντιμετωπίζουμε το πρόβλημα της αποτίμησης συνεχών ερωτημάτων σε μεγάλες ροές δεδομένων που αλλάζουν συχνά. Προς αυτή την κατεύθυνση, υιοθετούμε την HIFUN, μια γλώσσα ερωτημάτων υψηλού επιπέδου, που προτείνεται για την έκφραση αναλυτικών ερωτημάτων σε μεγάλα σύνολα δεδομένων. Η HIFUN προσφέρει ένα σαφή διαχωρισμό μεταξύ του εννοιολογικού επιπέδου, όπου τα αναλυτικά ερωτήματα ορίζονται ανεξάρτητα από τη φύση και τη θέση των δεδομένων, και το φυσικό επίπεδο όπου τα ερωτήματα αυτά αποτιμώνται, εκφράζοντας τα είτε ως MapReduce διαδικασίες είτε ως SQL ερωτήματα υποστηρίζοντας έτσι διαφορετικούς τύπους δεδομένων. Χρησιμοποιώντας τη HIFUN, σχεδιάζουμε έναν αλγόριθμο για την αυξητική αποτίμηση συνεχών ερωτημάτων, επεξεργάζοντάς μόνο το πιο πρόσφατο διαμέρισμα δεδομένων και εκμεταλλευόμενοι τις ήδη υπολογισμένες πληροφορίες, χωρίς να απαιτείται η αποτίμηση του ερωτήματος πάνω από το πλήρες σύνολο δεδομένων. Στη συνέχεια, μεταφράζουμε τον γενικό αλγόριθμο σε SQL και MapReduce χρησιμοποιώντας το SPARK, εκμεταλλεύοντας τις μεθόδους επανεγγραφής ερωτημάτων που παρέχονται από τη HIFUN. Χρησιμοποιώντας ένα συνθετικό σύνολο δεδομένων, επιδεικνύουμε την αποτελεσματικότητα της προσέγγισης μας στην επίτευξη της απόδοσης αποτίμησης της επερώτησης. Τέλος, αποδεικνύουμε ότι υιοθετώντας τις επίσημες μεθόδους επανεγγραφής επερωτήσεων της HIFUN, επιτυγχάνουμε την περαιτέρω μείωση του υπολογιστικού κόστους, προσθέτοντας άλλο ένα επίπεδο βελτιστοποίησης των ερωτημάτων στην υλοποίηση μας.

Ευχαριστίες

Η ολοκλήρωση αυτής της εργασίας επισφραγίζει την επιτυχή άφιξη στο τέλος μιας επίπονης, αλλά και συνάμα πλούσιας διαδρομής σε γνώσεις, εμπειρίες και δεξιότητες. Αρχικά, θα ήθελα να ευχαριστήσω τον Καθηγητή κ. Δημήτρη Πλεξουσάκη, για την εμπιστοσύνη αλλά και τις ευκαιρίες που μου έδωσε από τη περίοδο των προπτυχιακών μου σπουδών έως και σήμερα, με την ολοκλήρωση αυτής της εργασίας. Επίσης, τον Επίτιμο Καθηγητή κ. Νικόλαο Σπυράτο για την καθοδήγηση και την υποστήριξη καθ' όλη τη διάρκεια διεκπεραίωσης της παρούσας εργασίας. Ιδιαίτερα, θα ήθελα να ευχαριστήσω τον ερευνητή κ. Χαρίδημο Κονδυλάκη για την πολύτιμη και ιδιαίτερης σημασίας καθοδήγηση που έλαβα κατά την φάση ανάπτυξης αυτής της εργασίας. Χωρίς αυτόν, αυτή η εργασία δεν θα είχε ολοκληρωθεί. Επίσης, θα ήθελα να ευχαριστήσω το Ινστιτούτο Πληροφορικής (ICS) του Ιδρύματος Τεχνολογίας και Ερευνας (FORTH) και συγκεκριμένα το Εργαστήριο Πληροφοριακών Συστημάτων (ISL) για την υποστήριξη κατά την διάρκεια των προπτυχιακών και μεταπτυχιακών μου σπουδών. Ένα τελευταίο, αλλά εξίσου σημαντικό ευχαριστώ στην οικογένεια μου, που ήταν δίπλα μου όλα αυτά τα χρόνια.

Contents

List of Figures

Chapter 1

Introduction

Data emanating from high-speed streams is prevalent everywhere, in today's data eco-system. Example data streams that are rapidly updated, include IoT data [60, 14], network traffic data [58], financial tickers [66], health care transactions [50, 45, 34], the Linked Open Cloud [2, 3] and so on. In order to extract knowledge, find useful patterns, and act on information present in these streams, these data need to be rapidly analyzed and processed. However, this is a challenge, as new data arrive continuously at high speed, and efficient data processing algorithms are needed.

The research community has already provided open-source distributed batch processing systems like Hadoop [13] and MapReduce [21], that allow query processing over static and historical data sets, enabling scalable parallel analytics. Detailed surveys on MapReduce and Hadoop is available in [37, 40, 48, 51]. Actually, MapReduce has already been established as a framework for performing scalable parallel analytics and data mining on vast amount of data; and there is already a remarkable body of literature on MapReduce, but also some controversy mainly from the database community [24, 57]. MapReduce follows the functional programming model [52] and performs explicit synchronization across computation stages. The wide use of MapReduce is due to several reasons: it is offered as a free and open source implementation; it is easy to use [22, 47]; it is widely used by companies like a Google, Yahoo! And Facebook; and has been delivering excellent performance on extreme scale benchmarking [29, 33]. All these factors have fueled a rapid adoption of MapReduce for various types of data analysis and processing [26, 67, 46, 44, 19, 20].

In MapReduce, every job has the following cycle: first the input data are read, the processed, and finally written back to the Hadoop filesystem. Following jobs can consume the output produced, however is should be reread from the file system. As such, for iterative algorithms, that want to read once, and iterate the data many times, the MapReduce model introduces a significant overhead. To tackle this limitation, Spark [64] emerged on top of Hadoop, using the Resilient Distributed Datasets (RDDs) which implement in-memory data structures for caching intermediate data across a set of nodes. Since RDDs can be kept in main memory, the various algorithms can iterate over the RDD data efficiently. In nowadays, SPARK has gained impressive traction, with many additional advantages such as fault tolerance and efficient data processing exploiting main memory storage.

However, even with those technologies, processing and analyzing large volumes of data, in batch, is not efficient enough. This is true especially in scenarios that need rapid response to change over continuous (big) data streams [32]. Consequently, stream processing has gained significant attention. Several streaming engines including Spark Streaming [65], Spark Structured Streaming [7], Storm [30], Flink [17], and Google Data Flow [5], have been developed to that purpose. Continuous query processing, is a major challenge in a streaming content. A continuous query is a query which is evaluated automatically and periodically over a data set that changes over time [10, 59, 42, 41] and allows the user to retrieve new result from a data set without the need to issue the same query repeatedly. The results of continuous queries are usually fed to dashboards, in large enterprises, to provide support in the decision-making process [11, 27] etc.

1.1 Motivation and Contribution

As new data are constantly arriving at a high rate, the data sets grow rapidly and re-evaluation of the query incurs delays. Therefore the problem we focus in this thesis is *incremental query evaluation*, that is, given the answer of the query at time t, on data set D, how to find the answer of the query at time t' on data set D', assuming that the answer at time t has been saved and results become stale and stagnant over a time. Incremental processing is an auspicious approach for refreshing mining results as it uses previously saved results, to avoid the cost of re-computation from scratch. There is an obvious relationship between continuous queries and materialized views [28, 12, 35, 49, 68, 36], since a materialized view is a derived database relation whose contents are periodically updated by either a compete or incremental refresh based on a query. Incremental view maintenance methods [4] exploit differential algorithms to re-evaluate the view expression in order to enable the incremental update of materialized views. However, in our case, both the methods and the target are different.

In this thesis, we study this problem in the context of HIFUN, a recently proposed high level functional language of analytic queries [56, 55]. Two distinctive features of HIFUN are that (a) analytic queries and their answers are defined and studied in the abstract, independently of the structure and location of the data and (b) each HIFUN query can be mapped either as a SQL group-by query or as a Map-Reduce job. To summarize, at a high level, the main contributions of this master thesis are the following:

- We present a framework able to offer analytic information over a highly heterogeneous dataset, including both structured and unstructured data.

- We use the HIFUN language to define the continuous query problem in the abstract and give a high-level algorithm for its solution.
- We map this generic algorithm to the physical level, implementing the evaluation mechanism both as SQL queries and Map-Reduce tasks.
- We map the HIFUN query rewriting methods to the physical layer and we experimentally show that out implementation provides considerable benefits in terms of efficiency.

The experimental results indicate that, in terms of performance, our implementation is at least as good as the conventional ones, in most of the cases however, offering a significant advantage in query answering in terms of efficiency. To the best of our knowledge, our approach is unique in presenting incremental algorithms for both the high-level HIFUN language and the corresponding low-level mapping of those algorithms to the map-reduce and the group-by SQL models.

1.2 Thesis Outline

The remaining of the thesis is organized as follows. Related work is presented in Section 2. Then, the theoretical framework and the query language model used are presented in Section 3. More specifically, in Section 3.1 we present the semantics and internals of the HIFUN language, namely the notion of Analysis Context (subsection 3.1.1); the abstract definition of a query and its answer (subsection 3.1.2); the formal approach to rewriting HIFUN queries (subsection 3.1.3); and the adopted conceptual schema for query evaluation (subsection 3.1.4). In Section 3.2 we present a detailed description of how this conceptual schema can be applied over an evolving dataset using an incremental approach. In Section 4, we present a detailed description of the implemented system, we describe how an evolving dataset relates to micro-batching (subsection 4.1); and how the HIFUN conceptual evaluation scheme for continuous queries can be mapped to physical level mechanisms depending on the nature of data (subsections 4.2 and 4.3). In Section 5, we evaluate query evaluation, using the incremental computation instead of the baseline approach of batch computation. We also show the improvements exploiting the rewriting rules for reducing the evaluation cost. Finally, Section 6 concludes this thesis and present directions for future investigation.

Chapter 2

Related Work

In the literature, there are many studies already on collecting, storing and querying huge amounts of data both for static and dynamic datasets. In this thesis, we focus primarily on the problem of processing multiple continuous queries over evolving datasets. Many systems have been developed around this topic either in a centralized or in a distributed one.

As in this work we are focusing on continuous queries model and how these queries can be evaluated incrementally, in this chapter, we will initially present an overview of recent state of the art on continuous query processing. Then, we also present an overview of the functional models that exist in the literature, for the analysis of large volumes of transactional data sets.

2.1 Continuous Queries Based Approaches

The data stream and continuous queries problem has been extensively researched in both in the past and in recent years. For example in [9] the authors present models and issues in data stream issues. In this thesis, we focus on semantics for continuous queries and how those semantics mapped to an existing physical level mechanism.

2.1.1 Tapestry

Continuous queries were introduced as SQL- based language in Tapestry [59], named TQL, for content-based filtering over an append-only of email and posting massages database. Conceptually, a restricted subset of the SQL was used and it was converted into an incremental query that was defined to retrieve all answers obtained in an interval of t seconds. The incremental query was issued continuously, every t seconds, and the union of answers returned constituted the answer to the continuous query. An incremental evaluation approach was used, to avoid the repetitive computations and to return only the new results to the users. However, this approach was envisioned to append-only systems, as in our case, in which we suppose that the data set to be analyzed can only increase in size between successive time moments, an assumption common in data warehouses environments.

2.1.2 NiagaraCQ & OpenCQ

NiagaraCQ [18] and OpenCQ [42] use continuous queries over changing data, as a periodic execution of one-time queries as in Tapestry. NiagaraCQ is a distributed continuous query system that allows continuous XML-QL queries to be posed over dynamic Web content. The issue of scalability is addressed by grouping continuous queries for efficient evaluation. OpenCQ is another system focusing on continuous queries, for monitoring streaming web content. It focuses on scalable event-driven query processing and uses a query processing algorithm based on incremental view maintenance. In [63] the authors further discuss rate-based query optimization for streaming data in the context of NiagaraCQ. The similarity with our approach, is that NiagaraCQ and OpenCQ support incremental evaluation of continuous queries by considering only the changed portion of each updated source file and not the entire file. However, both systems focus on continuous queries over relational database sources, and thus do cannot handle unstructured streaming data.

2.1.3 Event-condition and publish-subscribe systems

Event-condition and publish-subscribe systems are also related. Event condition action methods [58] provide a mechanism to implementing event-driven querying in a conventional SQL database, by using continuous queries defined over special append-only active tables. Content-based filtering engines XFilter [6] and YFilter [25] perform efficient filtering of XML documents, based on user profiles, expressed as continuous queries using XPath [23] language. Their solutions, however, focused only on specific nature of data sets and designed as centralized systems.

2.1.4 COUGAR & TinyDB

Two other systems, COUGAR [14] and TinyDB [43] deal with query processing in sensor networks. In COUGAR, the authors define a data model and long-running queries semantics for sensor databases. A sensor database combines stored and sensor data. Stored data are represented as relations, while sensor data are represented as time series. Long-running queries are formulated using SQL with extensions and define a persistent view, which is updated at given time intervals. TinyDB is also a distributed query engine that runs on each of the nodes in a sensor network. Both systems, COUGAR and TinyDB are distributed query processors that run on sensor nodes with the TinyOS [38] operating system. Consequently, they are platform dependent.

2.1.5 AURORA & STREAM

AURORA [1], is a workflow-oriented system that allows users to build query plans by arranging operators, and the data flow among the operators, and then uses those specifications to determine how and when to shed load.

STREAM [10] is a framework that focuses on addressing the demands imposed by data streams on data management. The authors pay attention on memory management to enable approximates query answering. In particular, one of the project's goals is to understand how to efficiently run queries in a bounded amount of memory.

Both of these systems can process streaming data but they are designed as centralized systems. In this thesis, we propose a continuous query framework which utilizes state-of-the-art big data technologies.

2.2 Functional Query Language Models

In this subsection, we focus also on works related to functional models which can be used for the analysis of large volumes of detailed transaction data. A functional model was presented in [53] for data analysis in data warehouses over star schemas, using a definition of query similar to the one used in HIFUN. In [54], a language for data analysis was presented based entirely on partitions of the data set. Moreover, a notion of query rewriting was proposed based on the concept of quotient partition. However, no algorithms for query rewriting were presented. The functional query language FQL was presented in [15, 16], as an alternative to the relational model. The basic property of the FQL query language is the set of simple functional operations which can be combined using the function composition operation in a similar way to the HIFUN.

2.3 Conclusion

The above studies are primarily focusing on allowing users to query stored data. In order to make those approaches scalable, big data technologies are needed. Spark [64] and Spark Streaming [65] have been adopted by the industry as key technologies in developing big data systems. To this direction, Flink [17] was also proposed as a platform for processing of massive streams, and provides the ability to process distributed data. In general, both Spark and Flink aim to support most data processing workloads in a execution engine. The main difference is that respective architecture of each can prove limiting in certain scenarios. Spark Streaming divides streaming into discrete chunks of data called micro-batches and repeats the processing workload in a continuous loop. Instead of processing the streaming data one record at time, Spark Streaming discretizes the streaming data into tiny, sub-second micro-batches. This architecture, Spark Streaming's ability to batch data and leverage the Spark engine leads to higher throughput to other streaming systems.

Hence, we propose a framework which utilizes the Spark engine to evaluate incrementally continuous queries in order to analyze huge amounts of data distributed and independently of the nature of the data.

Chapter 3

The Query Language

In this section, we describe the HIFUN model [55, 56] and how this model applies over an evolving data set using an incremental approach. The model offers a clear separation between a conceptual and the physical level, which means that it can be used to define (and evaluate) analytic queries independent of the specific nature and location of the data sets (structured, unstructured, centrally stored or distributed). For more details on the HIFUN language the interested reader is referred to the relevant papers.

3.1 The Formal Model

3.1.1 Analysis Context

In our model, the context is an acyclic graph with a single root and a data set is an assignment of set functions, one to each arrow of the graph. More specific, the basic notion that the HIFUN model uses is the notion of *attribute* of a data set. An *attribute* is a function from the data set to some domain of values. In addition, as commonly implemented in practice, to analyze a data set, analysts use an analysis context, consisting of a number of different *attributes.*

As a running example consider a database in a distribution center, which collects and delivers products of various types in a number of branches. Figure 3.1 shows the analysis context of this data set D stored in the distribution center's database. The data that appears in an invoice has a unique identifier and shows the branch and the region in which the delivery took place, the date, the type of the product, the number and cost of units delivered. We define this information as a set of six attributes, namely b, r, d, p, q and cst. Following this perspective, given an invoice identifier, the attribute b returns the brunch, the attribute r returns the region, the attribute d returns the date, the attribute p returns product and the attribute q and cst returns the quantity and the cost of the product respectively. These is the primary characteristics of the data set, so the attributes with domain the data items of D, are called *direct attributes.* However, each of these characteristics determines one or more secondary characteristics of the data

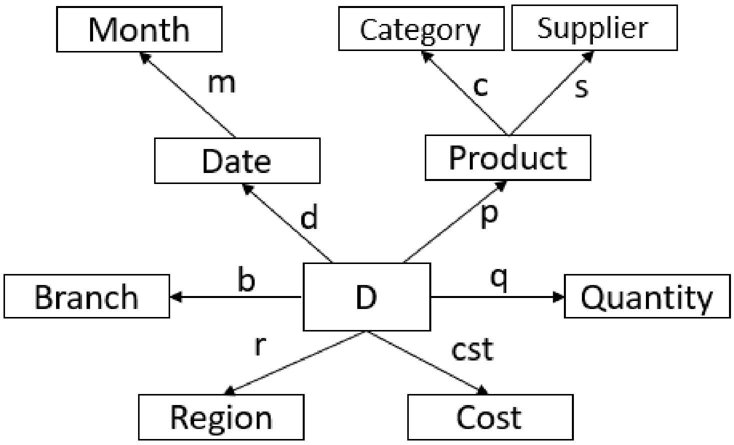

Figure 3.1: Analysis Context Example

set. For instance, as shown in Figure 3.1, the Data determinates the Month and the Product determines both Category and Supplier. Although these secondary attributes might not appear on the invoice, they can usually be inferred from the primary characteristics, and are useful for data analysis propose. These is the secondary characteristics, so the attributes that can be derived from the direct attributes are called *derived attributes.*

The functions represent information about some application being modelled. Combining these functions by using function algebra we can acquire new information about the application. More details about the combinations of these functions is available in following sections.

We note that, the context can have more than one root. That means that data analysis concerns two or more different data sets, possible of different nature and possible sharing one or more attributes. The study of these characteristics is out of the scope of this thesis.

3.1.2 Query Definition

A query is defined to be an ordered triple $Q = (g, m, op)$ such that g and m are functions of the context labeled as *grouping attributes* and *measuring attributes* respectively, and op is an aggregate operation that performs a calculation on a set of m-values. Formally, we have the following definition: let D be a finite set of data items, such that $D = \{d_1, ..., d_n\}$. An analytic HIFUN query over D is an ordered triple $Q = (g, m, op)$, where g is function with domain the set D and range a set A, m is a function with domain the set D and range a set V, and op is an operation over V taking its values in a set W. If $\{a_1, ..., a_n\}$ is a set containing the values of g over D (clearly $k \leqslant n$), then we call grouping of D by g, the partition $\pi_g = \{g^{-1}(a_1), ..., g^{-1}(a_k)\}$ inducted by g on D. The reduction of m with respect to op, denoted $red(m, op)$ is a value of W defined as $red(m, op) = op(< m(d_1), ..., m(d_n) >)$. On the basis of the above definitions, the answer to Q, denoted as ans_Q, is a function from a set of values of g to W defined by $ans_Q(a_i) = red(m/g^{-1}(a_i), op)$, $i = 1, ..., k$. Figure 3.2 shows the relationship between the function ans_Q and the functions appearing in the query Q.

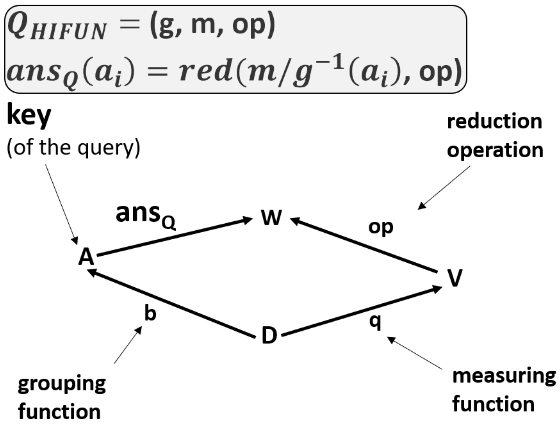

Figure 3.2: A query Q and its answer ans_Q

Restricted queries can also be defined over D. A restricted query is a query, which is attribute-restricted and/or result-restricted. An attributed-restricted query is defined as $Q = (g/E, m, op)$, where E is any subset of the domain of D. The evaluation of this type of query, requires the computation of restriction g/E and then the valuation of query $(g/E, m, op)$, over E. A result-restricted query is defined as $Q = (g, m, op)/F$, where D is any subset of the domain of definition of ans_Q. The evaluation of this type of query, requires the evaluation of (g, m, op), over D to obtain its answer ans_Q and then the computation of the restriction ans_Q/F.

Returning to our running example, assume that we want to know the total quantity delivered to each branch only for month 'December'. Formally, this query is written as $Q = (b/E, q, sum)$, where $E = \{x | x \in D \wedge (m \circ d)(x) =' December'\}$. This computation needs only three functions, namely b, q and $m \circ d$ among the set of functions that are defined in context of Figure 3.1. Figure 3.3 (a) illustrates an example of the data returned by b, q and $m \circ d$ and the computations needed during the query evaluation process. In order to find the total quantity by branch for month 'December', the following steps should be executed:

(a) *Grouping*: The grouping based on b/E creates a group for each branch which is different than the obtained when grouping is based on b. During this step, all invoices that happened in month 'October', referring to the same branch are grouped together.

(b) *Measuring*: In each group computed during the previous step, we find the quantity corresponding to each invoice by extracting the value using the function q.

(c) *Reduction*: For each group, we sum up the quantities. Then the relation of each branch to the corresponding total quantity is the evaluation of query Q, illustrated in 3.3(b).

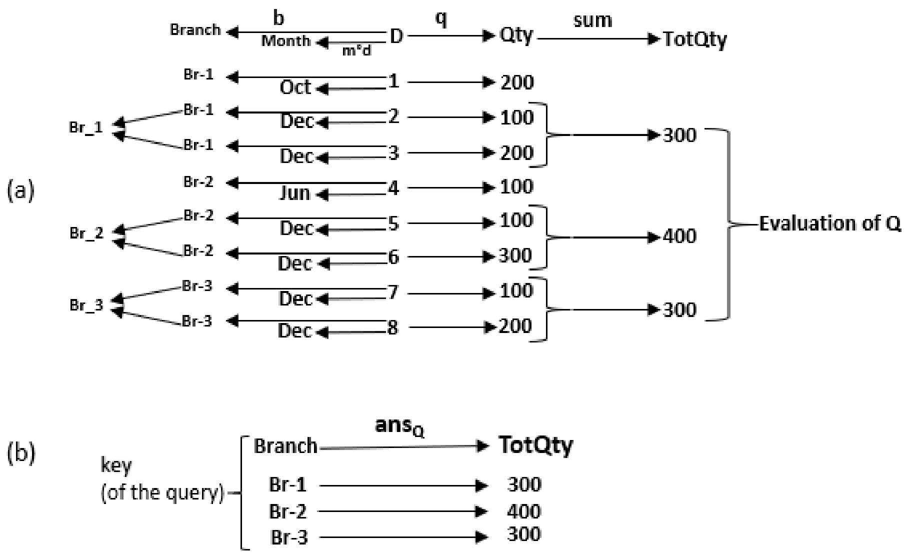

Figure 3.3: An analytic query and its answer

Furthermore, a defined query over context contains complex grouping functions using the following four operations on functions: *composition(*$\circ$*)*, *pairing(*$\wedge$*)*, *restriction(/)*. More specifically:

(a) *Composition*: This operation takes as input two functions f and g, and returns a function $g \circ f$. As mentioned above, a composition operation can be used to compose one or more attributes to support grouping by 'derived' attributes. For example, refer to context of the Figure 1, the following queries contains the composition operation on grouping part. (e.g. $(s \circ p, q, sum)$ or $(c \circ p, q, sum)$).

(b) *Restriction*: as detailed described previously, the restriction operation can be used to express restricted queries.

(c) *Cartesian product projection*: The Cartesian product projection operation is necessary in order to be able to reconstruct the arguments of a pairing and it is useful for query rewritings explained in the following sections.

(d) *Pairing*: This operation used to allow grouping by more than one attributes. To see an example of pairing usage, refer to context of the Figure 1 and consider the following query: $Q = (b \wedge r, q, sum)$. The answer of this query is a function, namely $ans_Q : Branch \times Region \longrightarrow TotQty$ associating each pairing (branch, region) with a total quantity. Put it differently, the query Q asks for the total quantities delivered by branch and region. The pairing operation can be extended to more than two functions in the conspicuous way.

3.1.3 Query Rewriting

In above sections we presented the definitions of our query language over an analysis context. However, independently how a query is evaluated, the formal model of HIFUN supports a query rewriting. An incoming query or a set of queries can be rewritten at the conceptual level, in terms of other queries. Query rewriting has been studied extensively [39] and it still active topic in areas such as the semantic web [62]. In this section we briefly describe the rewriting rules of our model to optimizing the evaluation of a query or a set of queries. This is done by rewriting an incoming set of queries in terms of the results of queries which have already been evaluated and the results stored (for example kept in main memory).

3.1.3.1 Common Grouping and Measuring Rewriting Rule

$Q = \{(g, m, op_1), ..., (g, m, op_n)\}$: The set of Q contains n queries, all having the same grouping function and the same measuring functions, but possible different reduction operations. In this case the rewriting of Q is the following: $Q' = (g, m, \{op_1, ..., op_n\})$, meaning that the grouping and the measuring is done only once and the n reductions operations are applicable to the results of measuring.

To see through an example how the common grouping and measuring rewriting rule works, consider the following set of queries on the context of Figure 1: $Q = \{(s \wedge p, q, min), (s \wedge p, q, max)\}$. This set of queries asking for the minimum and maximum quantity delivered by supplier. To optimizing the evaluation of the Q is done by rewriting the incoming set Q exploiting the similarity in grouping and measuring part as follows: $Q' = (s \circ p, q, \{min, max\})$. Therefore, grouping and reduction can be performed simultaneously for both min and max operations.

3.1.3.2 Common Grouping Rewriting Rule

$Q = \{(g, m_1, op_1), ..., (g, m_n, op_n)\}$: The set of Q contains n queries, all having the same grouping functions, but possible different measuring and reduction operations. In this case the rewriting of Q is the following: $Q' = \{g, (m_1, op_1), ..., (m_n, op_n)\}$, meaning that the grouping is done only once and the n measuring and reduction operations steps are applied to the results of grouping.

To see through an example how the common grouping rule works, consider the following set of queries on the context of Figure 1: $Q = \{p, q, sum), (p, cst, sum)\}$. This set of queries asking for the total quantity delivered and the total cost of each product. In this case, to evaluation of Q is done by rewriting the incoming set Q exploring the similarity in grouping part as follows: $Q' = (p, \{q, sum\}, \{cst, sum\})$. Hence, the common grouping operation can be performed once for two different measuring attributes.

3.1.3.3 Common Measuring and Operation Rewriting Rule

$Q = \{(g_1, m, op), ..., (g_n, m, op)\}$: The set of Q contains n queries, all having the same measuring and the same reduction operation, but possible different grouping functions. In this case the rewriting of Q is the following: $Q = \{(g_1 \wedge ... \wedge g_n, m, op), (proj_{G_1}, (g_1 \wedge ... \wedge g_n, m, op), op), ..., (proj_{G_N}, (g_1 \wedge ... \wedge g_n, m, op), op)\}$. In this point, we note that reduction operation is required to be distributive. The query Q can be answered directly, following the abstract definition of answer (grouping, measuring, reduction) for each one of the n queries. Also, the query Q can be answered indirectly by the execution of Q', if we first answered the base query $Q_b = (g_1 \wedge ... \wedge g_n, m, op)$ and then the projection queries are evaluated using the result from the base query which has already been evaluated and their result kept in main memory.

Let see an example for this rewriting rule. Suppose the context of the Figure 1 and we want to know the total quantity delivered for each branch and the total quantity for each product. These queries can be formally written as follows $Q = \{(b, q, sum), (p, q, sum)\}$. The set of Q can be answered directly follow the abstract definition of the answer. The answer of Q defined by the following functions: $ans_B : Branch \longrightarrow TotQty$ and $ans_P : Product \longrightarrow TotQty$. The incoming set Q can be also rewritten to probably reduce the evaluation cost as follows:

$$Q' = \{(b \wedge p, q, sum), (proj_B, (b \wedge p, q, sum), sum), (proj_P, (b \wedge p, q, sum), sum), sum)\}$$

However, Q can also be answered indirectly and equivalently as Q', if we know the totals by branch and product according to the function: $ans_{Q_b} : Branch \times Product \longrightarrow TotQty$, then all we need to do is to evaluate each projection query using the corresponding projection function as a grouping function and then the projection query evaluated follow the abstract definition of the answer (grouping, measuring, operation).

3.1.3.4 Basic Rewriting Rule

$Q = \{g_2 \circ g_1, m, op\}$: This rewriting rule based on the basic idea that a functional expression when used as a grouping function, can be equivalently rewritten to other expressions. In this case the rewriting of Q is the following: $Q' = \{(g_1, m, op), (g_2, (g_1, m, op), op)\}$ meaning that the base query $Q_b = (g_1, m, op)$ is evaluated first, and the result used to answer the rewritten query Q'. This observation leads to our basic rewriting rule for queries that have a common measuring function and operation but different grouping functions and require that the aggregate operation to be distributive.

To see intuitively how the basic rewriting rule works, consider the following queries on the context of Figure 1. The query $Q = (p, q, sum)$ asking the totals by product and the query $Q' = (c \circ p, q, sum)$ asking for the totals by category. Clearly, the query Q' can be answered directly, following the abstract definition of answer (i.e. by grouping, measuring and reduction). However, Q' can also be answered, if we know (a) the totals by product and (b) which products are in which category. Then all we have to do is to sum up the totals by product in each category to find the totals by category. Now, the totals by product are given by the answer to Q, and the association of products with categories is given by the function c. Therefore, the query Q' can be answered by the following query Q'', which uses the answer of Q as its measure: $Q'' = (c, ans_Q, sum)$, asking for the sum of product totals by category. Note that the query Q'' is well formed as c and ans_Q have Product as their (common) source.

3.1.4 Conceptual Query Evaluation Scheme

HIFUN offers a clear separation between the conceptual level, where analytic queries are defined and the physical level where analytic queries are evaluated. Using the batch processing approach, we first have to store the available data and then evaluate the query. In detail the following steps have to be followed:

(a) *Query Input Preparation.* $IN(Q)$ denotes the set of tuples which contain the information for evaluating query Q, independently of whether the data set is centrally or distributed stored. In this step k sets of tuples $I_1, ..., I_k$ are returned, that form a partition $\pi_{IN(Q)}$ of the input $IN(Q)$, where each tuple contains a data item identifier and the values of its attributes g and m, including the values of any possible attributes contained in the query restrictions.

(b) *Attribute Filtering.* If there are no attribute restrictions on query definition, this step is skipped. Elsewhere, filtering is performed on $IN(Q)$ tuples according to the query attribute restrictions.

(c) π_g *Construction.* This step constructs the partition $\pi_g = \{G_1, ..., G_n\}$, as it was previously defined in the query definition. The reduction of π_g will produce the answer to the query.

(d) π_g *Reduction.* Once the block G_j has been constructed, it can be reduced by the operation defined in the query definition, to obtain the answer on the value g_j of $g : ans_Q(g_i) = red(m/G_j, op)$.

(e) Result Filtering. If there are no result restrictions on query definition, this step is skipped. Elsewhere filtering is performed on ans_Q according the restriction on the query results.

In our running example, the query $Q = (b/E, q, sum)$, where $E = \{x|x \in D \wedge (m \circ d)(x) =' December'\}$ is mapped to the aforementioned conceptual schema as illustrated in Figure 3.4.

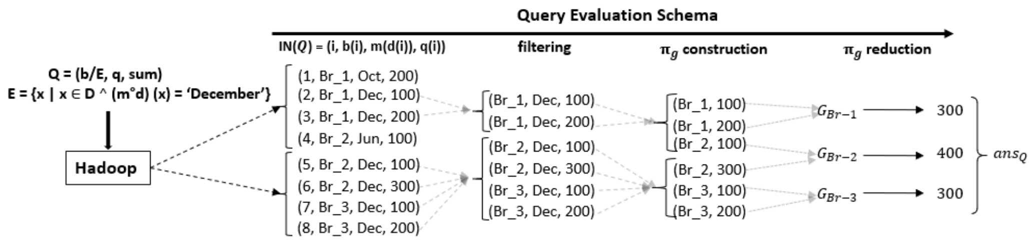

Figure 3.4: The conceptual scheme steps

3.2 Incremental Computation in HIFUN

In this section, we show how we can use the HIFUN language to incrementally evaluate continuous queries. An important common feature of real-life applications is that the input data continuously grow and old data remain intact. As such for the rest of this paper we assume that the data set being processed can only increase in size between t and t'. In such a scenario, the idea of incremental computation of a continuous query is to use the results of an already performed computation on old data and evaluate the query only on the lately appended data, merging eventually new and previous results.

Figure 3.5 illustrates our proposed incremental approach for continuous queries - the same query asked two times. We perceive the problem of incremental evaluation as follows: given the answer of a query Q at time t, on data set D, find the answer of the query at time t' on data set D', where $D' = D + \Delta D$, by evaluating the query only on ΔD and reusing the answer on D.

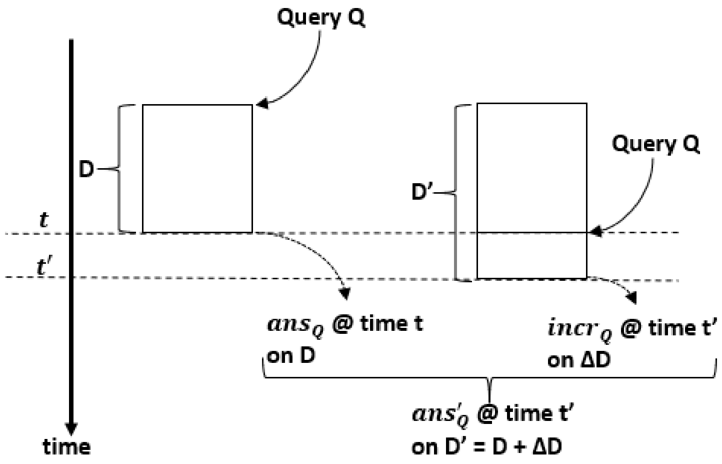

Figure 3.5: Incremental computing over append-only data set.

Now assume that the function ans_Q is the answer on D of Q at time t, including K groups of answers, and that the function $incr_Q$ is the answer on ΔD at time t', including the K' groups of answers. If the reduction operation op is a distributive operation, the answer ans' of query Q at time t', is evaluated as follows:

- **op=sum:** $ans'(i) = ans(i) + incr(i)$ if i is in $K \cap K'$; $ans(i)$ if i is in $K \setminus K'$; $incr(i)$ if i is in $K' \setminus K$;
- **op=min:** $ans'(i) = min(ans(i), incr(i))$ if i is in $K \cap K'$; $ans(i)$ if i is in $K \setminus K'$; $incr(i)$ if i is in $K' \setminus K$;
- **op=max:** $ans'(i) = max(ans(i), incr(i))$ if i is in $K \cap K'$; $ans(i)$ if i is in $K \setminus K'$; $incr(i)$ if i is in $K' \setminus K$;
- **op=count:** $ans'(i) = ans(i) + incr(i)$ if i is in $K \cap K'$; $ans(i)$ if i is in $K \setminus K'$; $incr(i)$ if i is in $K' \setminus K$;

Aggregate operations operate on a set of values to compute a single value as a result [60]. Distributive aggregate operations are those whose computation can be 'distributed' and be recombined using the distributed aggregates. All the operations that are previously described are distributive. This means that if the data are distributed into n sets, and we apply the aforementioned distributive operation to each one of them (resulting in n aggregate values), the total aggregate operation can be computed for all data by applying the aggregate operation for each subset and then combining the results. For example: $sum(1,2,3,4,5) = sum(sum(1,2), sum(3,4,5))$.

We also support non-distributive aggregate operations such as the average as: $avg(1,2,3,4,5) \neq avg(avg(1,2), avg(3,4,5))$. Non-distributive aggregate operations can be computed by algebraic functions that are obtained by applying a combination of distributive aggregate functions. For example, the average can be computed by summing a group of numbers and then dividing by the count of those numbers. Both, sum and count are distributive operations. More specifically:

- **op=avg:**
  - $ans'(i) = ans(i)$ if i is in $K \setminus K'$;
  - $ans'(i) = incr(i)$ if i is in $K' \setminus K$;
  - $ans'(i) = \frac{ans_{op=sum(i)+incr_{op=sum}(i)}}{ans_{op=count(i)+incr_{op=count}(i)}}$ if i is in $K \cap K'$;

Finally, there are additional aggregate operations, whose computation requires looking at all the data at once, and hence their evaluation cannot be decomposed into smaller pieces. Common examples of this type of aggregate operations include median and count-distinct. However, we leave those operations for future work.

Now consider the example illustrated in Figure 3.6. We would like to know the total quantity delivered to each branch during the month *December*. At time t the query was evaluated over the data set D, returning the function $ans_Q : Branch \longrightarrow TotQty$, as the answer of Q. Then, at time t' the query was again evaluated over only the data set ΔD, returning the function $incr_Q : Branch \longrightarrow TotQty$, as the answer of Q on D. In this case, the aggregate operation is the distributive operation sum. As such, we can produce the ans'_Q on time t' merging the functions ans_Q and $incr_Q$ as follows: The groups that appear only in K, which are the groups returned by the query Q at time t on D, are transferred directly to the result of ans'_Q. The groups that appear only in K', which are the groups of the query Q at time t' on ΔD, are transferred directly to the result of ans'_Q. The distributive operation sum is applied when the groups appear in the intersection of K and K'. For example, the key $Br-2$ appears in both K and K', therefore the answer ans'_Q for that key resulting as $sum(400 + 200) = 600$.

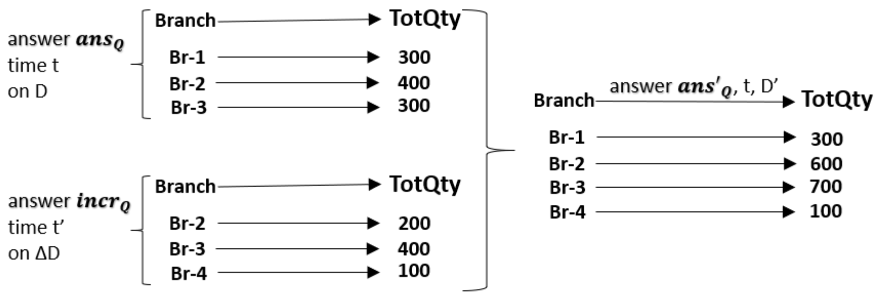

Figure 3.6: Incremental evaluation on our running example.

As already mentioned, HIFUN includes, out of the box, query rewriting rules. Using those is possible to reduce the evaluation cost. Assume for example the context of Figure 1 and the rewritten query $Q = (c, (p, q, sum), sum)$. Assume also that the rewritten query Q has already been evaluated on D at time t and the function $ans_Q : Category \longrightarrow Totals$ is the answer of Q. Figure 3.7 shows how we leverage the basic rewriting rule, to evaluate the query Q on ΔD at time t'. The rewriting rule requires the evaluation of the base query $Q_b = (p, m, sum)$ only on ΔD at time t'. The query Q_b is executed and the answer is returned as $ans_{Q_b} : Product \longrightarrow TotQty$. Therefore, the query Q can be answered on ΔD at time t' by evaluating the following query $Q' = (c, ans_{Q_b}, sum)$. The answer of the rewritten query Q', (the equivalent query of Q on ΔD) is computed by combining the function $incr'_Q$ on ΔD at time t' and the function ans_Q on D at time t as previously described.

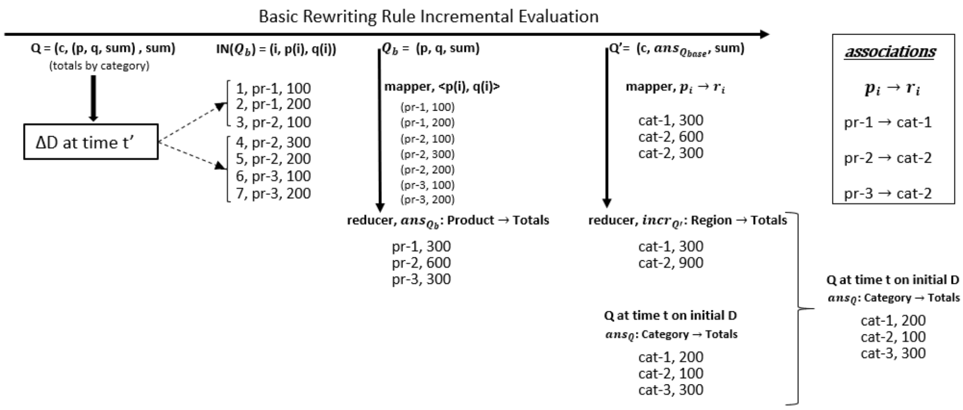

Figure 3.7: Example of basic rewriting rule.

Chapter 4

Implementation

As already shown, HIFUN queries can be defined at the conceptual level independent of the nature and the location of the data. These queries can be evaluated by encoding them either as map-reduce jobs or SQL group-by queries, depending on the nature of the available data. In this section, we show how to physically evaluate a HIFUN query processing live data streams. This is implemented using two different physical layer mechanisms: (1) the Spark Streaming [65] and (2) the Spark Structured Streaming [7]. Both mechanisms support the micro-batching concept - fragmentation of the stream as a sequence of small batch chunks of data. On small intervals, the incoming stream is packed to a chunk of data and is delivered to the system to be further processed [31]. This system based on definitions and features as formally proposed by HIFUN, and performs optimizations through incremental approach and query rewritings to reduce the computational costs.

4.1 Micro-batch stream processing

In the micro-batching approach, as a data set continuously grows and as new data become available, we process the tuples in discrete batches. The batches are processed according to a particular sequence. As a high volume of tuples can be processed per micro batch, the aforementioned mechanism uses parallelization to speed up data processing. An initial data set D_i is followed by a continuous stream of incremental batches ΔD_i that arrive at consecutively time intervals Δt. As we already explained, incremental evaluation would produce the query results at time $t + \Delta t$ by simply combing the query results at time t with the results from processing the incremental batches ΔD_i. Two key observations should be made here. The first is that computations needed are solely performed within the specific batch, following the evaluation scheme described in the previous section. Therefore, for every batch interval we calculate a result based on delta subset ΔD_i, e.g. $incr_i \longleftarrow e(\Delta D_i)$. The second observation is that a state should be kept across all batches. Stateful processing is able to handle unbounded streams of data. After the evaluation of each query is completed for each micro-batch, we need to keep the state across all batches. The previous state value and the current delta result are merged together and the system produces a new state incrementally, e.g. $state \longleftarrow u(incr_i, state)$. Figure 4.1, illustrates this incremental approach.

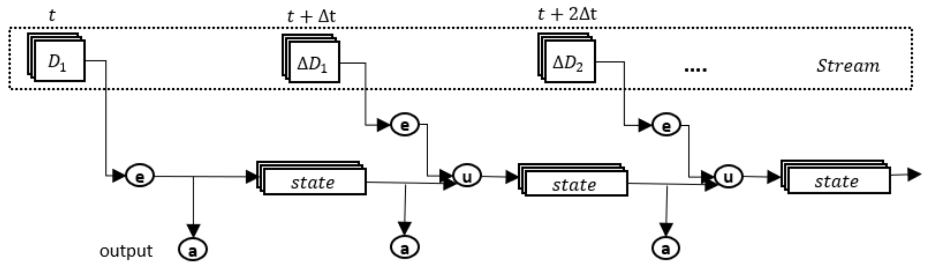

Figure 4.1: State maintenance.

4.2 Continuous HIFUN Queries to MapReduce

In [61] Glampedakis describes how the steps of the HIFUN conceptual evaluation scheme mapped to the existing physical level mechanisms of Apache Spark [64], using the Map-Reduce programming model and the Resilient Distributed Dataset (RDD), the main abstraction provided by Spark. In this work, the conceptual evaluation scheme is also implemented using the Map-Reduce programming model over the physical layer but exploiting Spark Streaming. Spark Streaming is a stream processing framework based on the concept of discretized streams and provides the *DStream API* which accepts sequences of data which arrive over time. The API implements the micro-batch stream processing approach with periodic checking of internal state at each batch interval. Internally, each DStream is represented as a sequence of data structure called Resilient Distributed Datasets (RDDs) which keeps the data in memory as they arrive in each batch interval. Batch interval also indicates how often an input RDD is generated. In the following, we describe in detailed how the conceptual schema presented previously is mapped to the physical layer mechanisms. Also, we describe the mechanism that allows the incremental algorithm to update continuous query results without recomputing them from scratch.

4.2.1 Conceptual Evaluation Schema to MapReduce

In this section we elaborate on the generic query evaluation schema, described in section 3, presenting details on its implementation over the physical layer when the Spark Streaming is used:

(a) *Query Input Preparation.* A set of attributes which are included in grouping and measuring part of Q is used to extract the information from the initial unstructured data set. In this step, the $IN(Q)$ set is computed and consists of tuples that contain the useful attributes values for each record. For this propose a map method is used to iterate through over all input records of the DStream and returns a new DStream which contains the information useful for the next evaluation steps.

(b) *Attributes filtering.* If attribute restrictions exist, this step filters the tuples of the DStream that do not conform to the query restrictions. The filter method applies on DStream and returns a new DStream containing only the elements that satisfy the queried predicate.

(c) π_g *construction*: To construct the grouping partition π_g, the map method is used. Each mapper receives the tuples to be used for extracting the key-values pairs from each data item. The result of this step - after the mapper which is applied to the tuples of DStream - is a new PairDStream which contains key-value pairs $< K, V >$. The key K is the value of the grouping attribute of each data item or the value of the grouping attributes if the domain of ans_Q is a cartesian product of two or more grouping attributes. The value V is the value of the measuring attribute of each data item.

(d) π_g *reduction.* In this step, each reducer uses the query operation op to reduce the set of key-value pairs received. The reduce-by-key method is applied and a new DStream is returned in which each RDD has a single element generated by reducing each RDD of the DStream.

(e) *Result filtering.* If result restrictions exist, this step filters the tuples of the DStream that do not conform to the query restrictions. The filter method is applied on DStream and the new DStream is returned containing only the elements that satisfy the queried predicates.

4.2.2 Rewritten Set Evaluation

As described in the formal definition of the query language, a set of Q can be rewritten according to some rules. In this section, we give a detailed description how the evaluation mechanism leverages these rules.

(a) *Common Grouping and Measuring Rewriting Rule.* In this case of rewriting, n number of different operations are applied to the common grouping and measuring attributes. In Query Input Preparation step extracted the information from the initial unstructured data set using the common grouping and measuring attributes which are appeared in the query set Q. For this propose a map method is used to iterate through over all input records of the initial DStream and returns a new DStream containing the values of the common grouping and measuring attributes for each record, useful for the next evaluation steps. To construct the grouping partition π_g, the previously generated $IN(Q)$ is iterated by a map method and a new pairing DStream created which contains the constructed key-value pairs $< K, V >$. The key

K is the value of the common grouping attribute for each DStream record and value V is synthetic and carrying a list of measuring attribute values. The measuring value of each DStream record is used and repeated n times to create a list of n values as a key K. The length n of the list is defined by the number of the operations which are appeared in the query set Q. The final π_g reduction is constructed when a reduction operation is completed. The reduce method applies for each key K the n operations on the list of values and produces the query answer in the form of key-value pairs $< K_q, V_k >$, where K_q is the key of the query and V_k its synthetic value containing the redacted value for each operation applicable to measure attribute. The answer of the Common Grouping and Measuring Rewriting Rule is completed when a set of $< K_q, V_k >$ is created.

(b) *Common Grouping Rewriting Rule.* The evaluation of this rewriting rule is slightly different to the evaluation of the Common Grouping and Measuring Rewriting Rule. In this case n number of different measuring attributes reduced to the common grouping attribute applying n possible different operations. In Query Input Preparation step extracted the information from the initial unstructured data set using the common grouping attribute and the n different measuring attributes which are appeared in the query set Q. A map method is used to iterate through over all input records of the initial DStream and returns a new DStream containing the value of the common grouping attribute and the values of n measuring attributes for each record, useful for the next evaluation steps. To construct the grouping partition π_g the previously generated $IN(Q)$ is iterated by a map method and a new pairing DStream is created which contains the constructed key-value pairs $< K, V >$. As we mentioned for the previous rewriting rule, the key K is the value of the common grouping attribute for each DStream record. The V in this rewriting rule is a synthetic value and carrying a list of n measuring attribute values. The length n of the list is defined by the number of the different measuring attributes which are appeared in the query set Q. The final π_g reduction is constructed when a reduction operation is completed. The reduce method applies for each key K the n operations on the list of n values and produces the query answer in the form of key-value pairs $< K_q, V_k >$, where K_q is the query key and V_k its synthetic value containing the redacted value for each operation applicable to measure attributes. The answer of the Common Grouping Rewriting Rule is completed when a set of $< K_q, V_k >$ is created.

(c) *Common Measuring and Operation Rewriting Rule.* In this rule one measuring attribute assigned to n different grouping attributes. The evaluation of the base query required first as follows. In Query Input Preparation step extracted the information from the initial unstructured data set using the common measuring attribute and the n different grouping attributes. A map method is used to iterate through over all input records of the initial DStream and returns a new DStream which contains the values of the n different grouping attributes and the value of common measuring attribute. To construct the grouping partition π_g for the base query, the previously generated $IN(Q)$ is iterated by a map method and a new pairing DStream is created which contains the constructed key-value pairs $< K, V >$. The key K is the value of the pairing operation applicable on n different grouping attributes of each DStream record and the value V is the value of the common measuring attribute of each DStream record. The π_g reduction for the base query is constructed using a reduce method which applies the common operation to redact the set of key-value pairs which are received. The intermediate result of base query has been produced in the form of key-value pairs $< K_{b_q}, V_{b_q k} >$, where K_{b_q} is the pairing key of the base query and $V_{b_q k}$ its redacted value. A set of key-value pairs is now available for the next evaluation steps. The set of key-value pairs $< K_{b_q}, V_{b_q k} >$ traversed n times to produce n new sets of key-value pairs. For each projection query a map-reduce job is needed to construct the answer as follows: a map method used to construct a set of key-value pairs $< K', V' >$. The key K' emitted as the value of the subset of the pairing key K_{b_q} which is specified by the projection operation. The value $V_{b_q k}$ emitted as a new $V' >$ value for the key $K' >$. The reduce method applied and produces the answer for the projection query in the form of key-value pairs $< K_q, V_k >$, where K_q is the query key and V_k its value. The answer of the Common Measuring and Operations Rewriting Rule is completed when n sets of $< K_q, V_k >$ are created.

(d) *Basic Rewriting Rule.* The evaluation of the base query required first in this rule. The Query Input Preparation step extracts the information from the initial unstructured data set using the grouping and measuring attribute which are appeared in the base query. The construction of the grouping partition π_g for the based query is needed. An iteration through the previously created $IN(Q)$ by a map method creates the new DStream which contains the constructed key-value pairs $< K, V >$. The key K is the value of the grouping attribute and V is the value of measuring attribute used in the base query of each DStream record. The π_g reduction for the base query is constructed using the reduce method which applies the operation to reduce the set of key-value pairs received. The intermediate result has been produced in the form of key-value pairs $< K_{b_q}, V_{b_q k} >$, where K_{b_q} is the base query key and $V_{b_q k}$ its value. A set of key-value pairs is available for the next evaluation step and used as follows. A map method is used to iterate through over all previously generated key-value pairs and constructs new set of key-value pairs $< K', V' >$ as follows. For each key K_{b_q} a new key K' emitted specified by the association between K_{b_q} and K' as defined in the context. The value V_{b_k} emitted as a new $V' >$ value for the key $K' >$. Final a reduce method applied and produces the final answer of this rewiring rule in the form of key-value pairs $< K_q, V_k >$, where K_q is the query key and V_k its value.

The deepest understanding of basic rewriting rule is coming through the following example. We assume the following HIFUN query $Q = (k, u, op)$, where k and u are the functions used by the mappers to extract the key-value pairs during the input preparation step and op is the operation applied by the reducers. If $k = g \circ f$ is the composition of two functions, then the query Q can be rewritten under the basic rewriting rule as follows $Q' = (g, (f, u, op), op)$. This implies that the initial query Q can be rewritten as a sequence of two other queries. The base query $Q_b = (f, u, op)$ should be executed first and the attributes f and u used during the Query Input Preparation step. The resulting query $Q' = (g, ans_{Q_b}, op)$ should then be executed, based on the previous result, as follows: the mapper used to construct the key-values pairs by using the association of f with g that is provided from the function g and then the reducer applies the reduction by the operation op on the set of constructed key-pairs.

Whichever rewriting method if is applied, the produced answer is a function or functions where the function has a domain of values to a set of values and represented as a set or sets of key-value pairs. For each function the domain of values is a set of keys, each of those correlated with the key of the query. The incremental algorithm examines the set of keys independently of whether those keys occurred after evaluating the original query Q or the rewritten one. In the next subsection, the details of the incremental evaluation are provided.

4.2.3 Incremental Evaluation

The aforementioned jobs are executed using Spark Streaming for each incoming micro-batch. When a query is executed, an answer is produced for a micro-batch and a DStream is created which encapsulates a key-value pair in the form of a $DStream[(K, V)]$, where K is the key of the continuous query that appears in the current micro-batch and V is the value of the reduction operation. We have to note that we maintain the state across the micro-batches (using the mapWithState method), using the key-value pairs produced for each micro-batch. Stateful transformation is a particular property used in this case and it enables us to maintain state between micro batches across a period of time, and it can be as long as an entire session of streaming jobs. That operation is able to execute partial updates for only the newly arrived keys in the current micro-batch. As such, computations are initiated only for the records that need to be updated. The state information is stored as a mapWithStateRDD, thus benefiting from the distribution's efficiency and effectiveness of Spark.

Let see now how the incremental update mechanism leverages the rewriting rules and allow to update the state or states between the micro-batches. We distinguish the incrementalization of rewritings in two cases.

Case 1. The first case includes the rules *Common Grouping and Measuring*

Rewriting Rule, *Common Grouping Rewriting Rule* and *Basic Rewriting Rule*. In these rules when a rewritten query is executed over a micro-batch a DStream is created in a form of $DStream[(K_q, V_k)]$. The K_q signifies the key of the query and V_k signifies the synthetic value of its K_q in the current micro-batch. The mapWithState method is used to update the current state, which is also in the form of $DStream[(K_s, V_{ks})]$, where K_s signifies the key of the aggregated query and V_{ks} signifies the synthetic value of its K_s. In each micro-batch, this method executed only for the keys of the state that needs to be updated, which is a great performance optimization.

Case 2. The second case includes the *Common Measuring and Operation Rewriting Rule*. In this rule when a rewritten query is executed over a micro-batch, n number of DStreams are created in a form of $DStream[(K_q, V_k)]$. Here, the K_q signifies the key of the projection query and V_k signifies the value of its K_q in the current micro-batch. The n DStreams depends on the number of different grouping attributes appear in the rewritten set Q. A chain of mapWithState methods are used to update the n current states, which are also in the form of $DStream[(K_s, V_{ks})]$. The K_s signifies the key of the aggregated projection query and V_{ks} signifies the value of its K_s.

4.3 Translating Continuous HIFUN Queries to SQL

In [61] Glampedakis describes how the HIFUN conceptual evaluation scheme implemented using the existing physical level mechanisms of Apache Spark SQL [8], which is a Spark module for structured data processing. In this work, we show how a query in HIFUN can be evaluated when the involving data set D is stored in an unbounded append-only relation table and also, we describe how we map the conceptual evaluation schema to the existing physical level mechanism using the semantics of the SQL exploiting group-by SQL queries of Spark Structured Streaming. The basic idea in Structured Streaming is treating continuously arriving data, as a table, that is being continuously appended. Structured Streaming runs in a micro batch execution model as well. Spark waits for a time interval and batches together all events that were received during that interval. The mapping mechanism defines a query on the input table, as if it was a static table, computing a result table that will be updated through the data stream. Spark automatically converts this batch-like query to a streaming execution plan. This is called instrumentalization: Spark figures out what needs to be maintained to update the result each time a new batch arrives. At each time interval, Spark checks for new rows in the input table and incrementally updates the result. As soon as a micro-batch execution is complete, the next batch is collected and the process is reapplied.

4.3.1 Conceptual Evaluation Schema to SQL

In [56] and [55]is already proved that HIFUN queries can be mapped to SQL group-by queries. In general, for the query $Q = (g_A, m_B, op)$, two cases are distinguished.

Case 1. The attributes A and B appears in the same table, say T. In this case we can obtain the answer of Q using the following group-by statement of SQL.

$$Select\ A, op(B)\ as\ ans_Q(A)\ From\ T\ GroupBy\ A$$

Case 2. The attributes A and B appear in two different tables, says S and T. In this case we can obtain the answer of Q using the following group-by statement of SQL.

$$Select\ A, op(B)\ as\ ans_Q(A)\ From\ join(T, S)\ GroupBy\ A$$

To this direction, let us see some examples of mapping analytic queries directly to SQL. We shall use the context of the Figure 4.2 and we shall assume that the data set is stored in the form of a relation data warehouse under the star schema shown in that figure. In general, a star schema includes one or more fact tables indexing any number of associated dimension tables. In our example, this star schema consists of the fact table FT and two-dimensional tables: the dimensional table DT_{Branch} of Branch and the dimensional table $DT_{Product}$ of the Product. The edges of the context are embedded in these three tables as functional dependencies that the tables must satisfy, and the underlined attribute in each of these three tables in the key of the table.

Our implementation handles the above relation schema as follows: the fact table represented as an unbounded table containing the primary incoming streaming data and the dimensional tables DT_{Branch} and $DT_{Product}$ are represented as static tables which are connected to the fact table. The assumption of the static dimensional tables is coming to avoid the stream-stream joins. The problem of generating inner join results between two data streams is that, at any time, the view of the data set is incomplete for both sides of the joining making it inefficient to find the matching values between two inputs data streams. Any row received from the input stream can match with any future not yet received row from the other input stream. Thus, the solution for this is coming, for both the input streams, by the buffering the past input as streaming state to match every future input with past input and accordingly generate join results. Since the above observations, our implementation has supported joins between a streaming and static relational table.

In this setting, consider the query $Q = (b/E, q, sum)$ where $E = \{x | x \in D \wedge d(x) =' 24/10/1992'\}$ over the context of Figure 10, asking for totals by branch in October 24, 2019. In this query, the grouping and measuring attributes appear in the same fact table. This query will be mapped to the following SQL query:

```
Select Branch, sum(Quantity) As ansQ(Branch)
From FT
Where Date = '24/10/2019'
Group by Branch
```

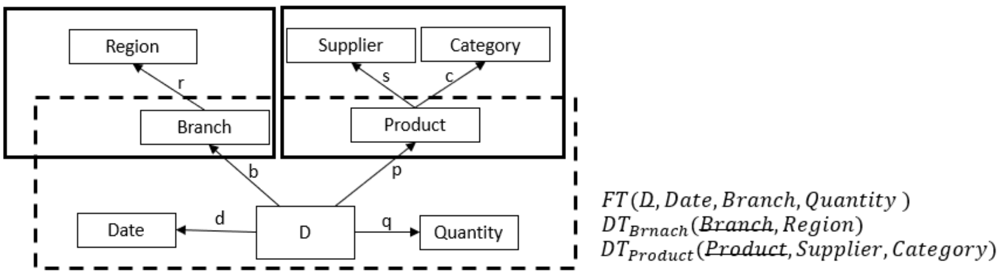

Figure 4.2: A context and its underlying data stored in the form of a relation schema.

Let see another example of a query evaluation step-by-step. Consider again the context in Figure 10 and suppose we need to evaluate the following query $Q = ((s \circ p) \times (c \circ p), q, sum)$ asking for the totals by supplier and category. In this query, the grouping attributes supplier and category appears in different table from the measuring attribute quantity. We map the Q to the following SQL query over a star schema:

Select Supplier , Category , **sum**(Quantity)
As $ans_Q(Supplier, Category)$
From join (FT, $DT_{Product}$)
Group by Supplier , Category

In Input Preparation Step the grouping attributes are selected which are the grouping attributes Supplier and Category and the measuring attribute Quantity. The attributes Supplier and Category appear in the dimensional table $DT_{Product}$ so the fact table FT and the dimensional table $DT_{Product}$ are joined accordingly. In the π_g construction step, the grouping partition as defined in the conceptual level is constructed using the 'Group by' clause that is used to group rows that have the same attributes Supplier and Category. In the π_g reduction step, is implemented by applying the query operation sum on the measuring attribute Quantity. The $'ans_Q(Supplier, Category)'$ is user defined attribute and the query returns the answer of Q in the form of a table with two attributes, $Supplier \times Category$ and $ans_Q(Supplier, Category)$.

4.3.2 Evaluation of the rewritten set

As mentioned before, a set Q of HIFUN queries can be rewritten according to some rules. Glampedakis [61] has already show how the rewritten set can be mapped to SQL group-by queries using the physical level mechanism of Spark SQL over a static relational tables. In this section, we give a detailed description of how the evaluation mechanism leverages these rules and a HIFUN rewritten set Q mapped to a physical level mechanism of Spark Structured Streaming and the semantics of SQL when the evolving data sets stored in an unbounded append-only relation table.

(a) *Common Grouping and Measuring Rewriting Rule.* In this rewriting rule, the SQL query is created customizing the π_g reduction step of SQL group-by query by adding the aggregate functions related to the n operations on the common measuring attribute which appears in the rewritten HIFUN Q set. Figure 4.3 shows the group-by SQL query decomposed into steps for this rewriting rule.

(b) *Common Grouping Rewriting Rule.* In this rewriting, the SQL query is created similarly as the previous rewriting rule. In the π_g reduction step of SQL group-by query adding the aggregate functions related to the n operations on the n corresponding measuring attributes which appears in the rewritten HIFUN Q set. Figure 4.4 shows how the rewritten set of HIFUN queries with common grouping attributes decomposed into steps and mapped to physical level SQL-group-by query.

(c) *Common Measuring and Operation Rewriting Rule.* This rewriting rule is not supported when the Spark Structured Streaming is used as physical level evaluation module. Firstly, a base table produced by the evaluation of the base query. In the next steps, this base table used for each projection query to produce the final result for the n grouping attributes appears in the rewritten set Q. The above computations are achievable under the SQL semantics by mapping the base HIFUN query to SQL-group-by query and each projection HIFUN query to projection SQL-group-by query. if the Spark Structured Streaming is used, the execution a chain of aggregation queries not supporting (until version 2.4.3)

(d) *Basic Rewriting Rule.* This rewriting rule is also not supported when the Spark Structured Streaming is used as physical level evaluation module. In this case, for the evaluation of the second HIFUN query, the answer table which is produced by the evaluation of the base query is joined with the table containing the grouping attribute of the second query. The above evaluation steps are achievable under the SQL semantics but a chain of aggregations queries required for this purpose. As mentioned before a chain of aggregation queries is not supported in Spark 2.4.3.

$$\boldsymbol{Q = (g_A, m_B, \{op_1, \ldots, op_n\})} \longrightarrow \begin{array}{l}\textbf{SELECT A, op}_{\mathbf{1}}\textbf{(B), \ldots, op}_{\mathbf{n}}\textbf{(B)}\\ \textbf{FROM T}\\ \textbf{GROUP BY A}\end{array}$$

Figure 4.3: The Common Grouping and Measuring Rewriting Rule to SQL group-by query.

$$\boldsymbol{Q = (g_A, \{m_B, op_1\}, \ldots, \{m_n, op_n\})} \longrightarrow \begin{array}{l}\textbf{SELECT A, } \boldsymbol{op_1(B), \ldots, op_n(Z)}\\ \textbf{FROM T}\\ \textbf{GROUP BY A}\end{array}$$

Figure 4.4: The Common Grouping Rewriting Rule to SQL group-by query.

Chapter 5

Evaluation

In this section, we describe the experiments that we conducted to evaluate our system. We expect that implementing an incremental query mechanism will result in a significant to the overall evaluation performance and scalability. In the following experiments, we compare our incremental approach with the batch processing approach to show the benefits that we can get form continuous queries when evaluated incrementally to avoiding unnecessary query evaluations. Also, we investigate the effectiveness of the query rewritings.

5.1 Data preparation

Our system was performed on a workstation cluster consisted of 4 nodes each equipped with 38 cores at 2.2 GHz, 250 GB RAM and storage capabilities of 1TB. On top Ubuntu LTS 16.04 was installed running Java version of 1.8.0.131 and Apache Spark 2.4.4. Spark was operated on top of Apache Mesos cluster with default configuration parameters for all experiments. For the data generation, we used a custom data generator to create a synthetic data size of 50GB split into 10 files of 5GB each(80M Records). Each dataset represented as an RDD and each RDD pushed into a queue and treated as a batch of data in the DStream, and processed like a stream. To distributed the data uniformly among all the cluster workers, the data follows uniform distribution. The following experiments were conducted over synthetic data sets stored in distributed file system (HDFS). In the case of the map reduce execution model the source data set is provided in a single text file and the analysis context of this data set is depicted in Figure 5.1, whereas in the case of SQL execution model, the source data set was structured according to relational table and the analysis context of this data set is depicted in Figure 5.2.

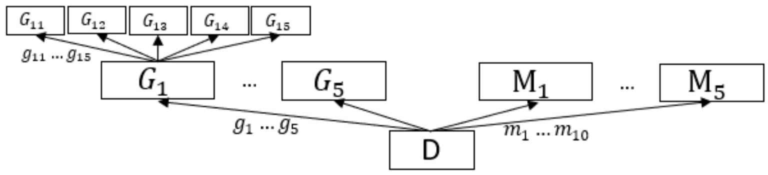

Figure 5.1: Analysis context of the unstructured data set.

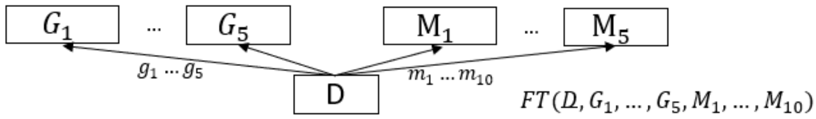

Figure 5.2: Analysis Context of the structured data set.

5.2 Continuous HIFUN Query Evaluation

In order to evaluate the effectiveness of the incremental evaluation of a HIFUN query against the base line approach, we define the following query $Q = (g_1, m, sum)$. Experiments started with an initial data set of 80M records. That data set was continuously growing over time and at each time interval, 80M new records were added to the existing data set. Using this data set, the batch computation approach looks at the entire data set when new data is available to be processed. The incremental approach on the other hand, only examines the new incoming data in the last time interval and incorporates the increment in the result. Figure 5.3 shows the performance of the two approaches when the HIFUN query is evaluated using MapReduce jobs or group-by SQL queries. The results show that the incremental approach shows a great benefit when used in practice: while the data set grows over a time, the evaluation cost remains stable independent of the overall increasing data size. In contrast, when the queries are evaluated over a batch data, the evaluation cost increases as the size of the input batch data increases as well.

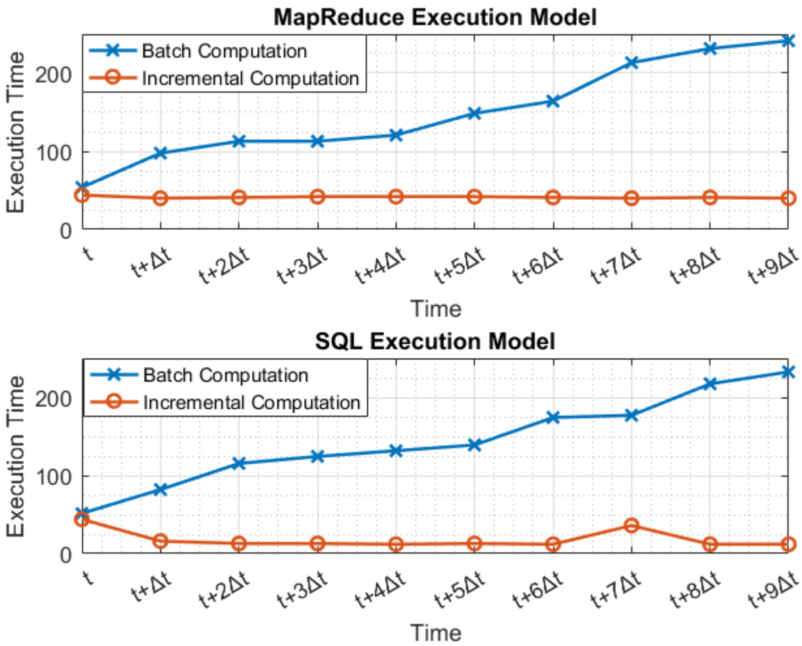

Figure 5.3: Evaluation of continuous HIFUN query

In the next set of experiments, a continuous HIFUN query is executed and the results produced by applying the incremental computations. We define the following two queries: $Q_1 = (g_1, m_1, sum)$ and $Q_2 = (g_1, m_1, avg)$. In both queries the same grouping and measuring attribute is used but different aggregation operations appear in those. We present the execution cost of each HIFUN query, contrasting efficiency and the aggregation operation which is used. As previously described, the non-distributive operations (e.g. avg) require the combination of synthetic computations to incorporate the increment in the result. The results show that the execution time of a HIFUN query is the same for both distributive and non-distributive operations. Figure 5.4 shows the performance when the HIFUN queries are evaluated using MapReduce jobs or group-by SQL queries.

Figure 5.4: Incremental evaluation of $Q_1 = (g_1, m_1, sum)$ and $Q_2 = (g_1, m_1, avg)$ using the MapReduce and SQL Execution model.

5.3 Common Grouping and Measuring Rewriting Rule Evaluation

In order to evaluate the *Common Grouping and Measuring Rewriting Rule* the grouping attribute g_1 and the measuring attribute m_1 used to create a set Q of 5 queries with 5 different aggregation operations applicable on measuring attribute m_1. The query set Q defined as follows:

$$Q = \{(g_1, m_1, sum), (g_1, m_1, min), (g_1, m_1, max), (g_1, m_1, count), (g_1, m_1, avg)\}$$

The equivalent rewritten of Q by this rule is the following query:

$$Q' = \{(g_1, m_1), (sum, min, max, count, avg)\}$$

In the first series of experiments for this rewriting rule, we evaluate the effectiveness of the incremental approach instead of a batch approach. The batch computation approach looks at the entire data set when new data is available to be processed. In this perspective, two different scenarios are evaluated: In the first scenario the Q executed by the evaluation of the included queries individually (e.g. without rewriting), and in the second scenario, the rewritten set Q' executed as defined by the rewritten theory. The incremental computation approach is more efficient by examines only the new incoming data in the last time interval and incorporate the increment in the result. In this perspective we evaluate again the two scenarios: the first scenario requires the execution of the Q; the second scenario requires the execution of rewritten Q'.

Figure 5.5 illustrates the evaluation time when Q and Q' are executed using the MapReduce execution model and the two different approaches; Figure 5.6 illustrates the evaluation time when Q and Q' are executed using the SQL execution model and the two different approaches. For example, at time $t + 3\Delta t$, the batch computation approach requires to execute the query set Q or the rewritten set Q', over all data generated in range of $t \leq +3\Delta t$. At time $t + 3\Delta t$, the incremental computation approach requires to execute the query set Q or the rewritten set Q', only on data generated in range of $t + 2\Delta t \leq t \leq t + 3\Delta t$ and then corporates the increment on the aggregated result. The experiment results prove the effectiveness of the theory about incremental computation. While a data set grows over a time, the evaluation cost remains stable independent of the overall increasing data size. When the queries are evaluated over a batch data, the evaluation cost growths linear accordingly of the size of the input batch data.

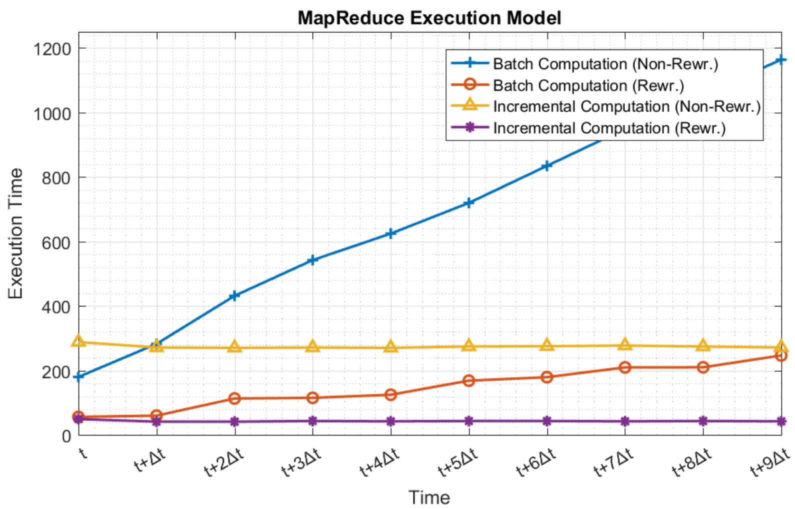

Figure 5.5: Evaluation of Common Grouping and Measuring Rewriting Rule when the MapReduce Execution model is used over an unstructured dataset.

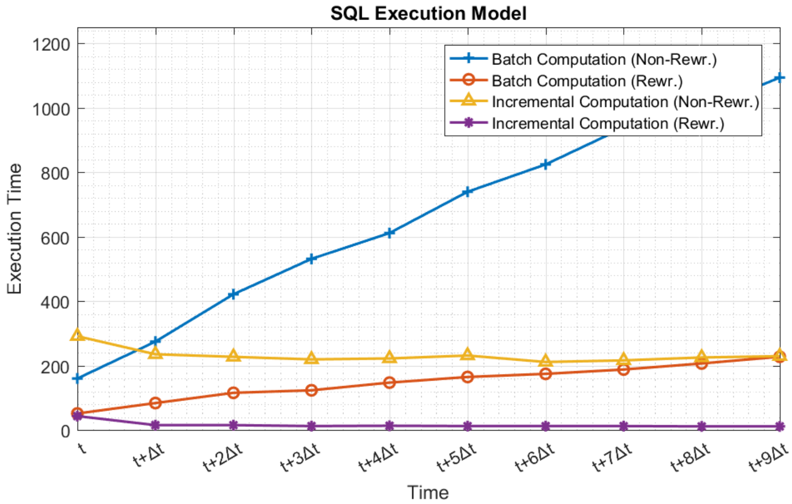

Figure 5.6: Evaluation of Common Grouping and Measuring Rewriting Rule when a SQL execution model is used over a structured dataset.

The next series of experiments evaluate the effectiveness of the *Common Grouping and Measuring Rewriting Rule* using the previously defined Q and Q', when the incremental processing approach used to refreshing previously generated results. Firstly, we evaluate the non-rewriting set Q by running the query evaluation process for a set Q of cardinality $n = 1$, and gradually increasing it to cardinality $n = 5$. In this scenario, each included query in Q executed for each micro-batch individually, and we report the average execution time as the average time of a set Q of cardinality n needs to executed incrementally for a specific number of incremental iterations over a synthetic data set. Secondly, we evaluate the rewriting set Q' by running the query evaluation for the rewriting set Q' of cardinality $n = 1$, and gradually increasing it to cardinality $n = 5$. In this scenario, each included query in rewriting set Q', executed for each micro-batch as defined by the rewriting theory and the average execution time is reported. Figure 5.7 illustrates the results of this series of experiments: when a non-rewriting query set Q is executed, the execution cost increase accordingly to the number of participating queries in the set. Moreover, we can be observed that the more queries participating in the rewriting set Q', the execution cost remains the same.

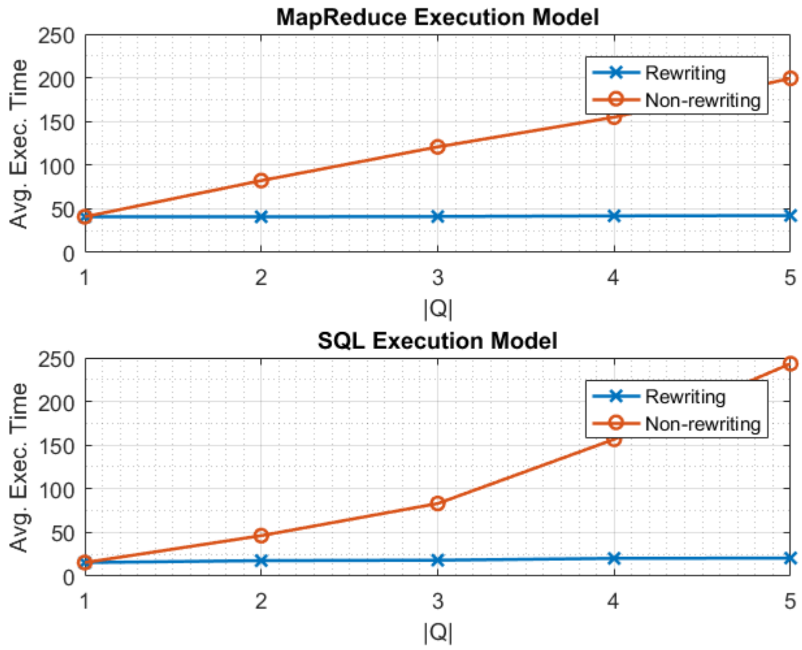

Figure 5.7: Evaluation of Common Grouping and Measuring Rewriting Rule for both, structured and unstructured datasets, while the cardinality of rewriting and non-rewriting set Q increases.

5.4 Common Grouping Rewriting Rule Evaluation

In order to evaluate the Common Grouping Rewriting Rule, the grouping attribute g_1 and five measuring attributes $m_1 ... m_5$ used to create a set Q of 5 queries with 5 different aggregation operations applicable on those measuring attributes. The query set Q defined as follows:

$$Q = \{(g_1, m_1, sum), (g_1, m_2, min), (g_1, m_3, max), (g_1, m_4, count), (g_1, m_5, avg)\}$$

The equivalent rewritten of Q by this rule is the following query:

$$Q' = \{g_1, (m_1, sum), (m_2, min), (m_3, max), (m_4, count), (m_4, avg)\}$$

In the first series of experiments the same evaluation experimental protocol as the previous subsection, is following. We evaluate the effectiveness of the incremental approach instead of a batch approach for this rewriting rule. Both approaches, batch and incremental approach, are evaluated in two different scenarios. Firstly, the included queries in Q evaluated individually and secondly, the included queries in Q evaluated as Q' as defined by the rewriting theory. Figure

5.8 illustrates the evaluation time when Q and Q' are executed using the MapReduce execution model and the two different approaches; Figure 5.9 illustrates the evaluation time when Q and Q' are executed using the SQL execution model and the two different approaches.

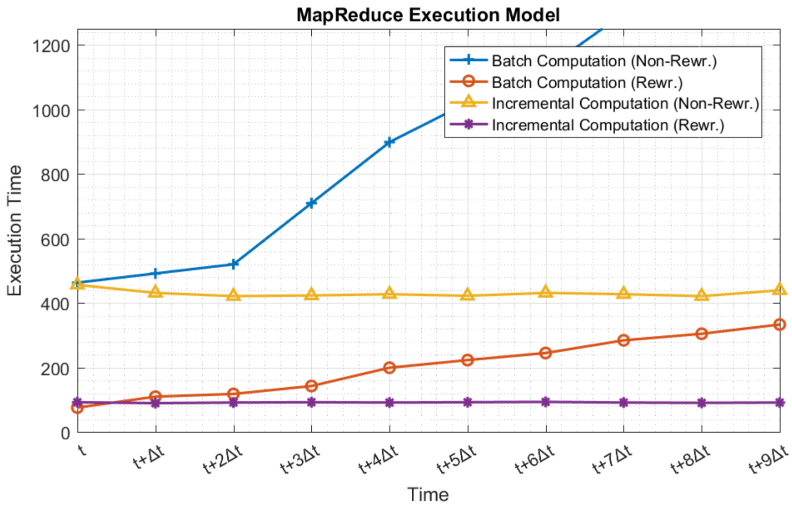

Figure 5.8: Evaluation of Common Grouping Rewriting Rule when a MapReduce Execution model is used over an unstructured data set.

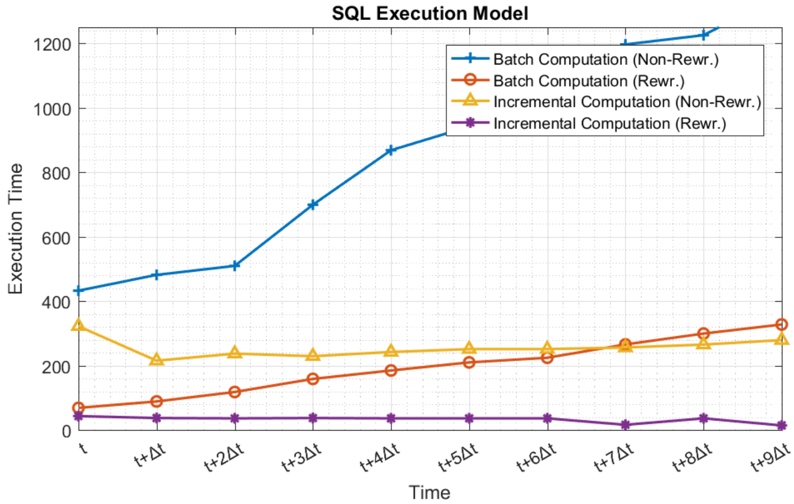

Figure 5.9: Evaluation of Common Grouping Rewriting Rule when a SQL execution model is used over a structured data set.

In the second series of experiments, we evaluate the effectiveness of the Common Grouping Rewriting Rule, using the defined Q and Q' when the incremental processing approach is used to incorporates the increment to the aggregated result. We investigate the evaluation time of Q and Q' of cardinality n=1 and gradually increasing it to cardinality n=5. Figure 5.10 illustrates the results of this series of experiments for both, structured and unstructured data sets.

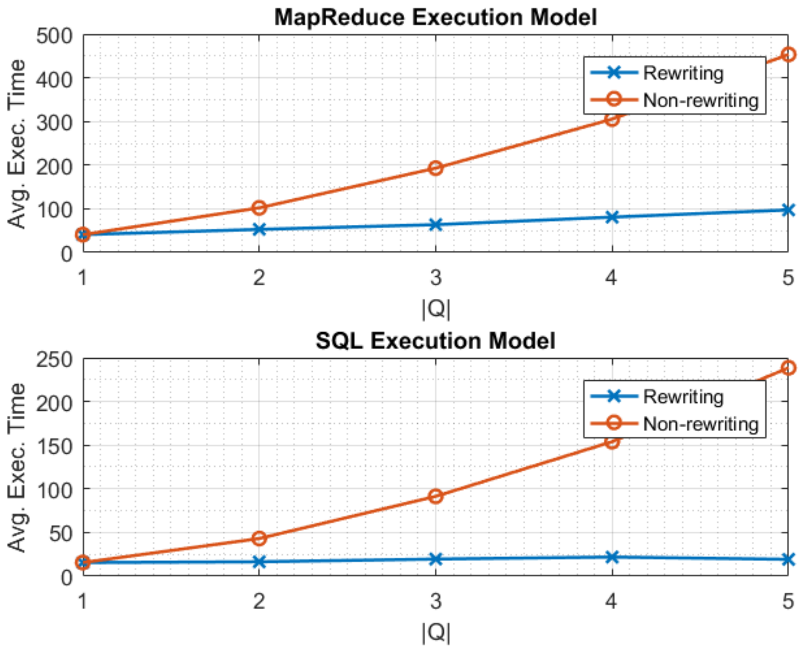

Figure 5.10: Evaluation of Common Grouping Rewriting Rule for both, structured and unstructured datasets, while the cardinality of rewriting and non-rewriting set Q increases.

5.5 Common Measuring and Operation Rewriting Rule Evaluation

In order to evaluate the *Common Measuring and Operation Rewriting Rule* the attributes g_1 and g_2 are used as grouping attributes, the attribute m_1 is used as measuring attribute and the aggregation operation sum applied on measuring attribute m_1.

$$Q = \{(g_1, m_1, sum), (g_2, m_1, sum)\}$$

The equivalent rewritten of Q by this rule is the following query:

$$Q' = \{(g_1 \wedge g_2, m_1, sum),$$
$$(proj_{G_1}, (g_1 \wedge g_2, m_1, sum), sum), (proj_{G_2}, (g_1 \wedge g_2, m_1, sum), sum)\}$$

We follow the experimental protocols that describe in the above subsections. Figure 5.11 illustrates the evaluation time when Q and Q' are executed using the MapReduce Execution Model for both approaches, batch and incremental computation.

In the second series of experiments in this rewriting rule, we define the set Q which is equivalent rewriting as Q' as formally described by the *Common Measuring and Operation Rewriting Rule theory.*

$$Q = \{(g_1, m_1, sum), (g_2, m_1, sum), (g_3, m_1, sum), (g_4, m_1, sum), (g_5, m_1, sum)\}$$

Figure 5.12 illustrates the evaluation time of Q and Q' while the cardinality of included queries in both sets increases from $n = 1$ to $n = 5$.

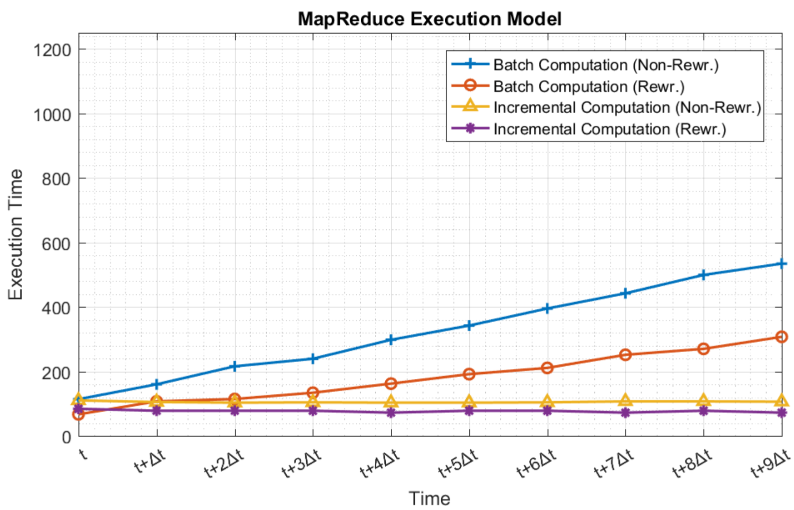

Figure 5.11: Evaluation of Common Measuring and Operation Rule when the MapReduce Execution Model is used over an unstructured data set.

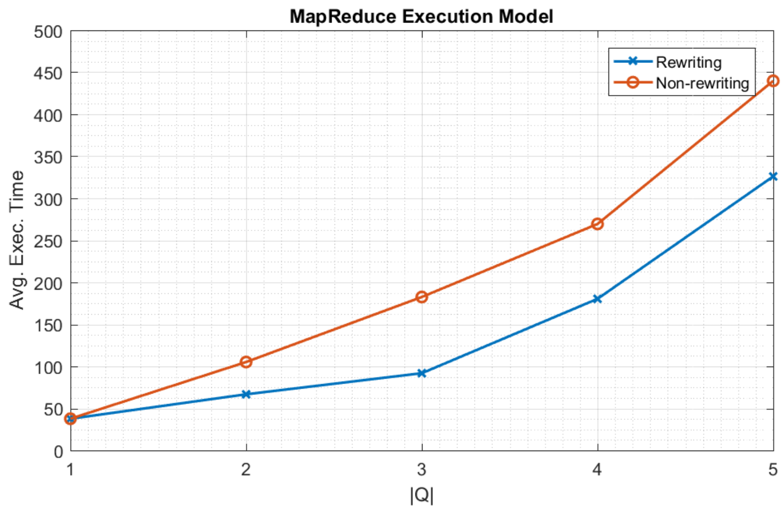

Figure 5.12: Evaluation of Common Measuring and Operation Rule for unstructured data set, while the cardinality of rewriting and non-rewriting query set Q increases.

5.6 Basic Rewriting Rule Evaluation

The first experiment that we conducted to evaluate the efficiency of the basic rewriting rule is following described. The continuous query $Q = (g_{11} \circ g_1, m_1, sum)$ and the rewritten continuous query $Q' = (g_{11}, (g_1, m_1, sum), sum)$ are defined, were both evaluated using map-reduce over an increasing synthetic data set using the context of Figure 5.13. In batch approach the non-rewriting query Q and the equivalent rewriting query Q' is evaluated. Furthermore, the incremental approach is used to evaluate both, Q and Q'. Analyzing the results of this experiments, we notice that when the incremental approach is applied instead of batch approach, the evaluation cost is redacted. The rewriting of $(g \circ f, m, op)$ not effects the computation cost, for both approaches, when we unfold the initial query to two other queries as describe above.

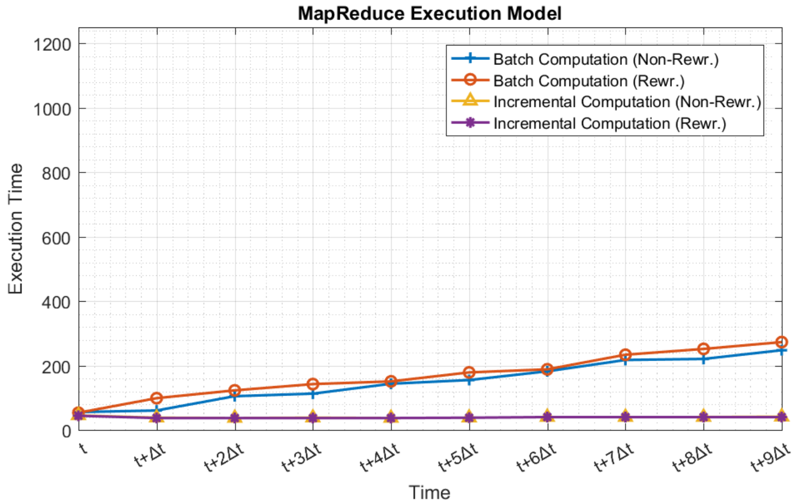

Figure 5.13: Evaluation of Basic Rewriting Rule when the MapReduce Execution Model is used over an unstructured data set.

Now we present another experiment on the unstructured data set by the context of Figure 12. We define the following set of queries:

$$Q = (g_{11} \circ g_1, m_1, sum), ..., (g_{15} \circ g_1, m_1, sum)$$

containing five queries and all of them have the same distributive operation applicable on the same measuring attribute m_1. As described by the rewriting theory, the Q can equivalent rewritten by the basic rewriting rule as follows:

$$Q' = \{(g_{11}, (g_1, m, op), sum), ..., (g_{15}, (g_1, m, op), sum)\}$$

The rewritten set Q' consists of five queries and each one uses the answer of (g_1, m, op) as its measure. To investigate the effectiveness of this rewriting rule, we run the experiments for a set Q' of cardinality $n = 1$ and increasingly the cardinality increments up to $n = 5$. We notice how the effectiveness of the basic rewriting rule adjusts as more queries participate in the rewritten Q'. Figure 5.14 shows the average evaluation time of the non-rewriting set Q and the rewritten set Q' accordingly to the number of participated queries. The result shows that the average evaluation time it is not affected while the number of participated queries increasing.

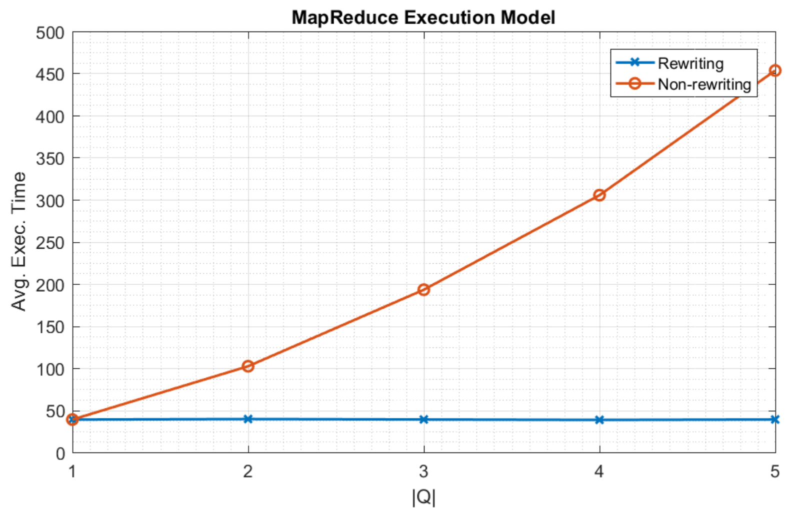

Figure 5.14: Evaluation of Basic Rewriting Rule while the cardinality of rewriting and non-rewriting set Q increases.

Chapter 6

Conclusion and Future Work

In this thesis, we leverage the HIFUN language, adding an incremental evaluation mechanism using Spark Streaming. We present an approach allowing the incremental update of continuous query results, preventing the costly re-computation from scratch. We also show the additional benefits of query rewriting, enabled by the adoption of the HIFUN language. The query rewriting rules can be implemented in the physical layer as well, further benefiting the efficiency of query answering. We demonstrated experimentally the considerable advantages gained by using the incremental evaluation, reducing the overall evaluation cost using both the map-reduce implementation and the SQL one. Our system provides a compact solution for big data analytics and can be extended to support a big variety of data set formats, with its evaluation mechanisms working regardless of the nature of the data.

Future work will exploit a number of research items that can be used for the extension of our system. The first concerns the evaluation of a query in millisecond low-latency processing mode of streaming called continuous mode. Our implementation mechanism has been providing stream processing capabilities through micro-batching. The main disadvantage of this approach is that each task (e.g. micro-batch) needed to be collected and scheduled at regular intervals, through which the minimum latency that the physical level module could provide. Suppose now we want to analyze fraudulent credit card transactions. Ideally, we want to identify and reject a fraudulent transaction as soon as the culprit has swiped the credit card. The continuous processing mode, instead of launching periodic tasks, attempts to overcome this limitation to provide stream processing with very low latencies.

The second research item concerns event time support in query evaluation. Event time is the time that each individual event occurred on its producing phase (e.g. generated from IoT device) and can be included within the record before enter in processing phase, and that event timestamp can be extracted from each record. When all the data has arrived, event time processing is able to produce correct and consistent results even when working with out-of-order or late events.

Bibliography

[1] Daniel J. Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alexander Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, Rui Yan, and Stanley B. Zdonik. Aurora: a data stream management system. In *SIGMOD '03*, 2003.

[2] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Incremental data partitioning of rdf data in spark. In *ESWC*, 2018.

[3] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. Rdf query answering using apache spark: Review and assessment. *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 54–59, 2018.

[4] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5:968–979, 2012.

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, F. Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8:1792–1803, 2015.

[6] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB*, 2000.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *SIGMOD '18*, 2018.

[8] Michael Armbrust, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD '15*, 2015.

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02*, 2002.

[10] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30:109–120, 2001.

[11] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Developer dashboards: The need for qualitative analytics. *IEEE Software*, 30:46–52, 2013.

[12] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD '86*, 1986.

[13] Chrisje R. Bolt. Hadoop: The definitive guide. 2014.

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

[15] Peter Buneman and Robert E. Frankel. Fql: a functional query language. In *SIGMOD '79*, 1979.

[16] Peter Buneman, Robert E. Frankel, and Rishiyur S. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7:164–186, 1982.

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[18] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD '00*, 2000.

[19] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[20] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, 2010.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *CACM*, 2004.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53:72–77, 2010.

[23] Steven J. DeRose and I. Corp. Xml path language (xpath) version 1.0. 1999.

[24] Janine DeWitt and Michael Stonebraker. Mapreduce: A major step backwards. 2014.

[25] Yanlei Diao, P Fischer, and Michael J. Franklin. Yfilter: Efficient and scalable of xml document. 2002.

[26] Vasilis Efthymiou, Petros Zervoudakis, Kostas Stefanidis, and Dimitris Plexousakis. Group recommendations in mapreduce. 2017.

[27] Amy Franklin, Swaroop Gantela, Salsawit Shifarraw, Todd R. Johnson, David J. Robinson, Brent R. King, Amit M. Mehta, Charles L. Maddow, Nathan R. Hoot, Vickie Nguyen, Adriana Rubio, Jiajie Zhang, and Nnaemeka G. Okafor. Dashboard visualizations: Supporting real-time throughput decision-making. *Journal of biomedical informatics*, 71:211–221, 2017.

[28] Ashish Gupta and Inderpal Singh Mumick. Materialized views: techniques, implementations, and applications. 1999.

[29] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, 2010.

[30] Muhammad Hussain Muhammad Iqbal and Tariq Rahim Soomro. Big data analysis: Apache storm perspective. 2015.

[31] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys Tutorials*, 17:381–404, 2011.

[32] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[33] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin Gyu Kim, Hyungsoo Jung, and Heon Young Yeom. Mrbench: A benchmark for mapreduce framework. *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18, 2008.

[34] Hian Chye Koh and Gerald Tan. Data mining applications in healthcare. *Journal of healthcare information management : JHIM*, 19 2:64–72, 2005.

[35] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *SIGMOD Conference*, 2010.

[36] Dominique Laurent, Jens Lechtenbörger, Nicolas Spyratos, and Gottfried Vossen. Monotonic complements for independent data warehouses. *The VLDB Journal*, 10:295–315, 2001.

[37] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Record*, 40:11–20, 2011.

[38] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, Jason L. Hill, Matt Welsh, Eric A. Brewer, and David E. Culler. Tinyos: An operating system for sensor networks. 2005.

[39] Alon Yitzchak Levy. Answering queries using views: A survey. In *VLDB 1995*, 1995.

[40] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46:31:1–31:42, 2014.

[41] Ling Liu, Calton Pu, Roger S. Barga, and Tong Zhou. Differential evaluation of continual queries. *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 458–465, 1996.

[42] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng.*, 11:610–628, 1999.

[43] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, 2005.

[44] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, 2010.

[45] Gunasekaran Manogaran, Daphne Lopez, Chandu Thota, Kaja Abbas, Saumyadipta Pyne, and Revathi Sundarasekar. Big data analytics in healthcare internet of things. 2017.

[46] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD '11*, 2011.

[47] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, 2009.

[48] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman, and Fabio Kon. A comprehensive view of hadoop research - a systematic literature review. *J. Network and Computer Applications*, 46:1–25, 2014.

[49] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 3:337–341, 1991.

[50] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. In *Health Inf. Sci. Syst.*, 2014.

[51] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ArXiv*, abs/1302.2966, 2013.

[52] Peter Sestoft. Analysis and efficient implementation of functional programs. 1991.

[53] Nicolas Spyratos. A functional model for data analysis. In *FQAS*, 2006.

[54] Nicolas Spyratos and Tsuyoshi Sugibuchi. Parallelism and rewriting for big data processing. In *ISIP*, 2012.

[55] Nicolas Spyratos and Tsuyoshi Sugibuchi. A high level query language for big data analytics. 2014.

[56] Nicolas Spyratos and Tsuyoshi Sugibuchi. Hifun - a high level functional query language for big data analytics. *Journal of Intelligent Information Systems*, 51:529–555, 2018.

[57] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53:64–71, 2010.

[58] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Annual Technical Conference*, 1998.

[59] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *SIGMOD '92*, 1992.

[60] Chun-Wei Tsai, Chin-Feng Lai, Ming-Chao Chiang, and Laurence Tianruo Yang. Data mining for internet of things: A survey. *IEEE Communications Surveys Tutorials*, 16:77–97, 2014.

[61] Glampedakis Vassilis. A big data analytics system based on a high-level query language using apache spark. 2017.

[62] Maria-Esther Vidal, Louiqa Raschid, Natalia Marquez, Marelis Cardenas, and Yao Wu. Query rewriting in the semantic web7. *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, pages 7–7, 2006.

[63] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*, 2002.

[64] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[65] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP '13*, 2013.

[66] Dongsong Zhang and Lina Zhou. Discovering golden nuggets: data mining in financial application. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 34:513–522, 2004.

[67] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *CloudCom*, 2009.

[68] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD '95*, 1995.