D.2.1 – Query Based Organization : Notions and definitions

Philippe Lamarre

HAL Id: hal-01174205
https://hal.science/hal-01174205

Submitted on 8 Jul 2015

D.2.1 – Query Based Organization : Notions and definitions

Philippe.Lamarre@insa-lyon.fr

1 Context and objectives.

Plug computers enable any of us to maintain cheap nano-clusters of Raspberry Pis at home, and to host data and services. These plug-based systems offer a middle-ground between traditional data-centers and P2P networks. In contrast to data-centers, plug-based systems rely on autonomous participants. This clearly puts back the control of our digital lives in the users' hands and reduces the asymmetric relationship many of us maintain with our service providers. Differently from P2P network, plug-based systems provide a stable infrastructure that exhibits little churn, and opens the possibility of novel and optimized mechanisms.

In this deliverable, we focus on users' queries. They have been shown to be a valuable resource, so that democratizing their access should support the flourishing of new creative activities. Without opposition to solutions brought by major companies, we aim at an alternative and complementary proposal to existing systems which opens the ability to enhance users' data to many more, while respecting their privacy.

Nowadays, the data produced in a continuous flow becomes increasingly important. Examples include information dissemination (news agencies, newspapers . . .), social networks (twitter and similar), location-based systems (smartphones, communicating GPS), sensors (buildings, vehicles, infrastructure. . .) etc. Even data produced on the Internet can be considered as continuous. This is why we focus on continuous queries. Classical services supported by major companies, such as Google Alerts [www], enable to query the data streams the service has itself selected. More recently, services based on connected objects enable compagnies to collect data which are very close to the users, see for example Google Nest [www]. These data are not published by companies, but this does not mean that the user has direct control on the use of collected data nor can he query the streams by himself. Indeed, it is actually really difficult for an user to get and to query the data streams he or his connected objects produce.

Enabling users to compute their queries on the data streams they choose, without having to communicate these streams or their queries to third party would be a major step. Of course, the objective is not to replace existing services but rather to simplify their participation to the system to take advantage of them. The computation of issued queries may rely on existing services, within the limits of what service providers accept.

We propose to explore a solution where the computation of queries is supported by users' resources. The objective is to allow an user to locally compute queries involving his private data and to control who he shares these data with, including his own queries and results. This is an interesting point, but the system would be very limited if restricted to local data and computation. Our objective is also to enable numerous users to get together to compute complex queries that are far beyond their individual capacities. Indeed, we have to keep in mind that a plug capacity is limited. With such limited resources, a group of users faces a situation similar in nature to the one experimented by scientists faced to a data volume too high to be handled by their individual resources.

The general idea is to handle continuous queries results in a peer-to-peer way. The users may ask which queries are handled by the system. Thus users interested in similar queries can cooperate within a community to compute them as well as to store and disseminate their results. An user running a query can also take advantage of results already obtained by others. Cascading query results from one user to another is what

we call a Query Torrent or shortly *QTor*. We will first focus on such a torrent over continuous queries and streams.

Expected advantages are numerous. First, access to users' queries is symmetric. Second, the query language is not fixed nor limited by a central actor, users are autonomous w.r.t. this aspect. Third, many users may gather together to handle costly computations which would be out of reach of an isolated user. Fourth, users may benefit of results obtained by others. Fifth, each user keeps control of her resources: as data, resources can exclusively be used for purposes which have been explicitly authorized. We are convinced this latter point is an important feature to ensure the users' adoption of the system. Sixth, resources availability naturally fits the needs: the more popular a query is, the more resources are devoted to it, the less expensive its computation is for an interested user and the more its results are replicated and available. Finally, optimization and scalability are improved: each query is computed only one time and its results are shared among interested users.

Without any restriction on the expressiveness of the querying language, we find it very interesting to pay a particular attention to privacy preserving aggregation queries. Here our objective is to make possible the private data mining via aggregation queries while avoiding privacy issue.

In this task, our two main objectives are: i) to define a generic organization that enables a community of users to share and capitalize on their queries and provided results, and ii) to permit the use of aggregation queries preserving privacy even if related to private data.

2   State of the art.

2.1   A typology of queries.

Data management requires storing, structuring, consulting, updating, deleting and other operations. To specify them, an user or an application developer can rely on an API or a declarative query language. We consider the use of a query language.

Query languages have several common points. First of all, usually they enable an user to express what s/he wants without having to know how to obtain it, *i.e.* they are declarative. This is of real interest for the user, and it greatly simplifies the development of applications. The query computation task is fully ensured by the data management system which has to provide services with highest efficiency. This has led to many works in many contexts, from a single query computation in a centralized system to the optimization of multiple queries in a distributed and parallel architecture. We distinguish several types of queries by considering three dimensions : frequency, complexity, popularity.

2.1.1   Query frequency.

Queries can be divided into three different classes: one shot queries, recurrent queries and continuous queries.

One shot queries are submitted to the system once. Historically, they have been studied first. They enable to manipulate a data collection with specific operations, such as selection, join, aggregation... They play a central role in data management and have received much attention from the scientific community and major companies these last decades. Depending on the kind of data and application, many queries languages have been developed. The best known is certainly SQL[Cod70] which applies on relational data, with an associated algebra. Semi-structured data like XML are most often queried through languages such as XQuery [w4w] or XPath [w5w]..., while RDF data is queried using SPARQL [w6w].

Recurring, or persistent, queries can be seen as one shot queries which are run periodically on a data set that can change from one run to another. This is typically the case for backups and data warehousing tasks such as loads and extracts, which may be run every night for example. Despite the apparent simplicity, this periodicity may lead to difficulties and possible optimizations. To manage this type of queries, classical database systems often include a scheduler [w11w]. Recently, the problem has been considered in the context of big data [LZRE15].

Continuous queries are devoted to data that take the form a possibly infinite continuous data streams, such as produced by sensors, GPS, connected objects as well as RSS streams, emails, tweets, etc. As the result of a continuous query is itself a data stream, it is possible to define an algebra of such queries. Using

filtering queries over data streams is an interesting approach, powerful enough to solve many concrete problems. Publish/Subscribe [KT13, EFGK03] systems are mainly based on such kind of queries and have many applications. PubSub languages generally provide boolean expressions of filters over manipulated data, which includes range queries. The main objective is to select specific elements of a data stream. To express aggregative and join queries over data streams, it is necessary to build more sophisticated languages, in particular by introducing the notion of windows over data streams. For example, CQL [BW01, ABW06] extends SQL in this way. C-SparkQL [BBC$^+$09] proposes a similar extension for SPARQL. In this field, a query takes a set of data streams as input and produces a data stream.

2.1.2   Query complexity

Depending on the expressiveness of the language, the complexity of queries may vary a lot, in terms both of the data volume and of the number of operations involved. Boolean queries and filters involving comparison of some attributes are rather simple queries, while join or range queries are more complex, in particular when queries are continuous. The more complex are the queries, the more important is the optimization of their processing. Query processing may consider a single query only, and optimize its execution plan, using statistics and other information. Multi-query processing seeks to use the processing of some queries in order to optimize the processing of others. This concept is used for range queries [GS04, SHLB08] and query containment [KV98, MS04, JK82]. In our view, query rewriting is a more general approach [Hal01, YP04, AD98, CDGL00] that fits our goal to design a generic proposal.

The problem of rewriting a query with respect to existing materialized views , "Answering Queries using Views" [Hal01], has been largely studied. In the field of relational databases, Rachel Pottinger and Alon Halevy have proposed in [PPHH01] the minicon algorithm which works for conjunctive queries. Next, it has been adapted and generalized to many other languages and contexts. Such algorithms have received a significant attention because of their relevance to many contexts and problems. For example, in data integration context, it is necessary to compute all possible rewritings of a query with respect to existing sources to obtain a complete result. In the case of the query optimization, the problem is to find one rewriting which is equivalent to the initial query but with a more efficient execution. Our context is similar to this latter case: the problem is to take advantage of existing materialized views to improve performance, not to enumerate all possible rewritings. Indeed, the query on a data stream also produces a data stream that other queries can use. Hence, it is possible to rewrite the query taking advantage of other queries. For example, the results of the query which selects football news for an user may be interesting for another query to select the news of a particular football team issued by an other user only interested in his favorite team. An interesting feature of the rewriting technique is that it is not limited to one to one relation over queries. It can also deal with more complex many to one, rewriting one query taking advantage of the combination of many queries.

2.1.3   Query popularity.

In all query systems, from DBMS to internet search engines, some queries are submitted by more users than others. An appropriate treatment of this phenomenon can afford a large resource saving. This is especially true in the treatment of continuous queries which, over time, can individually consume a large amount of resources. Interestingly, in this field, popularity is not reflected in terms of frequency, but in the number of simultaneous queries which may be easier to handle.

Query frequency, complexity and popularity are three dimensions that impact the volume of resources used to process the queries. Even a rather simple continuous query may require a lot of resources if it is very popular.

2.2   Query processing in distributed environments.

2.2.1   Peer-to-Peer as a model of distributed environment

A distributed environment is characterized by the presence of several, possibly many, sites that host information sources and users. The Peer-to-Peer (P2P) paradigm, usually introduced by Shirkey's definition [Ora01] considers P2P as *a class of applications that takes advantage of resources (storage, cycles, content,*

human presence) available at the edges of the Internet. P2P systems [AH01, AH02] have emerged in the 2000's to solve scalability issues. They had a lot of success for file-sharing applications [PGES05, KB$^+$05] but have also been developed for distributed computing [And04] or search engine [BMT$^+$05], data management [NOTZ03, TIM$^+$03], collaborative platform [MMGC02, KBC$^+$00]. They also have been used to scale up Publish/Subscribe systems [KT13].

To ensure high scalability and to prevent bottlenecks and single points of failure in the network, an ideal P2P system is a self-organizing system composed of equivalent nodes without using any centralized information. Thus, all nodes have the functionalities to client and server in the same time and all resources are shared and directly accessed by the others nodes. Moreover, the network is able to evolve according to successive peer connections or disconnections (*churn*).

The SocioPlug project considers federated and distributed plugs having low resources. Each plug can be both a source of data streams (server) and an user (client). In this regard, the P2P paradigm fits our context very well and will be used in this whole section as a generic model. Notice that, when applied to plugs, some hypotheses such as high churn may be relaxed.

Two main categories of P2P architectures have emerged: *unstructured P2P* and *structured P2P* architectures. There are several types of unstructured P2P architectures: *(i) centralized P2P* makes use of a specialized node which acts like a central server, *(ii) pure P2P* where the P2P paradigm is completely respected and *(iii) hybrid P2P* where the most efficient peers (according to their capacities like cpu or bandwidth), called Super-Peer, organize themselves to play the role of a central server. To improve the scalability of a P2P system, peers can be organized into overlays [LCP$^+$05] enabling to consider logical links between peers independently of the physical network of peers.

A structured P2P architecture is based on Distributed HashTable (DHT): Hash functions are applied on peers and data to organize them according to some data structure such as a ring for Chord [SMK$^+$01], a d-space for CAN [RFH$^+$01] or a tree for P-Grid [ACMD$^+$03]. The P2P system is then able to store and retrieve pairs of the form (key, values).

According to the same principles for common distributed query processing, querying a P2P system requires to retrieve data on which the query is executed in order to produce results that is returned to the query sender. This result can be stored or directly exploited to execute another task. Each step is specifically defined according to the considered P2P architecture. Many challenges, especially about location performance, have been considered since the 2000s.

For processing a one-shot query in unstructured P2P architectures, the data retrieving is initially based on message flooding propagated on the P2P network. The query is encapsulated on messages and can be executed on each peer receiving the message. To limit the number of opened connections the result is forwarded to according to the reversed path of query propagation. Location improvements can be obtained as an alternative way for message flooding by considering aggregative index for routing message like the routing indices [CGM02], by introducing a semantic level on peers and their connexions like in SON [CGM05], or by introducing gossip protocols by selecting the neighbour through which the message will be propagated. Another way to improve data location process is to exploit data replication [MPV06].

Structured P2P architectures have been designed to improve data location process. The DHT structure enables to efficiently resolve the keyword-based query. Thus, the retrieving and execution steps are highly linked. Moreover, as one peer (or few peers, when replication is considered) stores, the result can be straightly sent to the peer of the end user.

2.2.2   Processing of Continuous Queries in Distributed Environments

In distributed systems, querying always raises two related problems: query evaluation and results dissemination. Query evaluation involves both computing the results (which may range from simple filtering to processing joins or range queries) and identifying the recipients to which the results have to be sent. These points may become critical considering numerous continuous queries over data streams. To improve the efficiency of the system, complementary directions may be explored. For example, some solutions focus on optimizing query evaluation (e.g. using multi-query plan optimization inspired from the database domain). Others are centered on result dissemination (e.g. organize the system according to a dissemination tree).

P2P-diet [IKT04] is an example of publish/subscribe super-peer architecture where the client peers pub-

lish the meta-data about their resources and post their continuous queries at some super-peer. The continuous query is then broadcasted to the super-peer backbone and replicated at each super-peer. Only subsumption relations between queries are considered.

Some distributed publish/subscribe systems, with brokers such as SemCast [PC05], or deployed over all the participants in a full peer-to-peer organization such as DPS [AGD+06], deploy complex organizations having several diffusion tree for the different predicates or several indexing levels. A Distributed Hash Table (DHT) may be used to store the continuous queries. In [KYS+06], each node is responsible of a list of queries and a conjunctive query being stored at the node responsible for one of its terms, which eases optimization by considering subsumption. DHTrie [TIKR14] uses an extended DHT to register subscriptions, which highly depends on the chosen structure (query language, data schema). This results in organizations that are hard to adapt, while the use of rewritings allows a generic organization that can efficiently be adapted.

Some other distributed systems, like CDNs [TB05] or Multicast-inspired systems (for instance, Split-Stream [CDK+03]), are based on the possibility for some participants to obtain all the data produced by the source, and to re-send them unchanged to the others. Such approaches lead to huge, redundant processing, as each participant has to work by its own, but may be used to design efficient diffusion scheme inside a QTor community.

A lot of propositions deal with local multi-query optimizations. For instance, in NiagaraCQ [CDTW00], grouping the query by their "signatures" based on the physical operators allow to limit the I/O. In RoSeS [TATV11], [CATV11], a local query graph is optimized by factorization. Those kind of propositions alone do not cope with huge number of participants, which requires a network-wide organization, but may efficiently be used in a QTor system to decrease the processing load of participants that contribute to several communities.

To the best of our knowledge, our approach is closest to [CF05] and Delta [KKM13]. SPO [CF05] is a very dynamic and incremental solution, but the diffusion tree is limited to containment and the solution does not address the participants' capacity limitations. Delta [KKM13] is based on query rewriting which is more general. However, Delta has a global, non incremental, system reorganization with high organizational and functional costs without decisive advantage on latency. Furthermore, it is not adapted to popular queries at all. Both systems do not take full advantage of powerful sources/participants and may overwhelm sources which is a problem we want to address. In addition, we think that the notion of query-based community introduces an intermediate abstraction which should favor scalability in presence of popular queries.

2.3 Massively parallel frameworks.

2.3.1 Query processing : distributed execution plan and massively parallel approaches

Distributed query processing is an important challenge for several years. With the evolution of data volumes, systems must be able to process complex queries on large amount of data. In order to scale-up systems throughput, the potential query parallelism must be exploited. Two main paradigms (Figure 1) deals with this issue : distributed execution plan and massively parallel approaches.

Distributed execution plan   Considering a SQL query with SELECT-FROM-WHERE clauses, it appears that this query can be divided into operators (selection, projection, join...) as presented in [YC84]. After this splitting phase, a query can be represented as a direct acyclic graph (DAG) where each vertex is an atomic operator. Depending on the query, some vertices can be executed simultaneously and on different processing units.

MapReduce paradigm   On an other hand, the MapReduce paradigm is a programming model detached from relational queries. Actually, inputs are considered as key/value pairs computed by two user defined functions : map and reduce. Those functions are defined as follow in [DG04]

- **Map** (k1, v1) $\longrightarrow$ list(k2, v2)

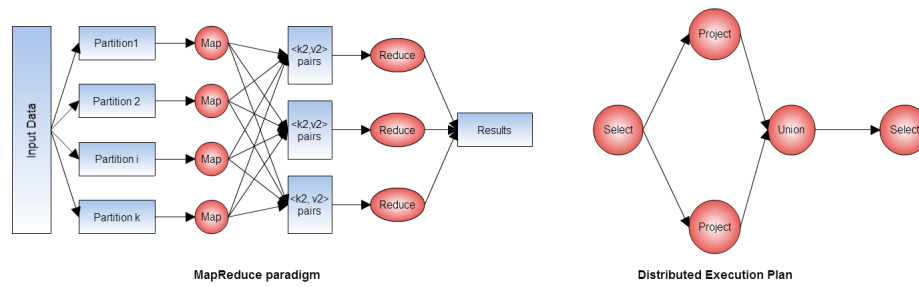- **Reduce** (k2, list(v2)) $\longrightarrow$ list(v2)

Figure 1: Distributed execution plan and MapReduce paradigms

After a data partitioning step, Map function is applied on each partition. Map function takes as input a key/value pair and produces a set of intermediate key/value pairs. All values associated to the same key are grouped and then given to Reduce function as input. This function takes an intermediate key and the set of associated values and merge them depending on user definition.

In addition to these paradigms, we identify three main axes as shown on Figure 2 to classify existing systems we considered : paradigm for query execution, query type and the data granularity.
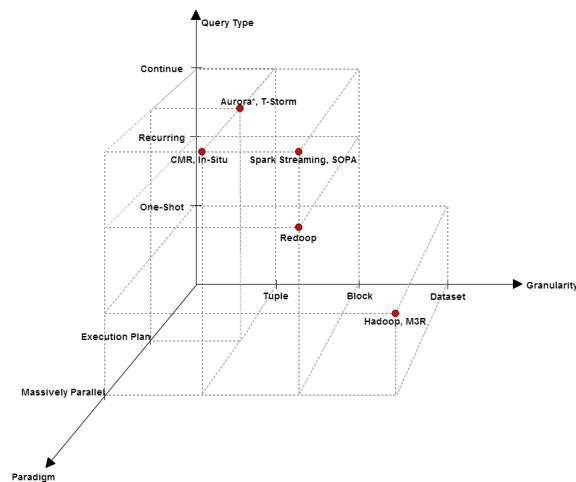

Figure 2: DSMS features

In next parts, we present how query type and data granularity influences query execution mechanisms and main issues they involve.

2.3.2 One-shot query processing

One-shot queries are defined on an entire dataset stored on disk and load for process afterwards. Solutions for one-shot query processing are from both paradigms but requires different optimizations.

Existing systems   On one hand, distributed execution plan are used by relational DBMS like [10] or NoSQL DBMS like [9] in order to share query execution load on a cluster of machines. Those systems balance load by estimating the cost of each operator and produce a equal distribution among processing units. On the other hand, MapReduce implementations like [8] include their own file system to manage datasets. Users can set the number of map and/or reduce nodes. This setting directly influences inputs partitioning and distribution among Map nodes. Some pre-defined distribution strategies are available like Round-Robin distribution. Other implementations add some constraints on the partitioning or the distribution strategy as presented in [SCSH12].

2.3.3 Recurring query processing

Nevertheless, many applications require results on a dataset with a smaller granularity. Considering a dataset composed by logs, it is relevant to get intermediate results for example by timestamp spans. The same query is then applied on sub-datasets so this query can be seen as a recurring query according to [LZRE14]. Contrary to one-shot queries which are executed once on a full-loaded dataset, recurring queries are long-lived periodic queries executed on an incrementally loaded dataset. This incremental aspect is crucial because it allows a loading rate based optimization to balance execution load among processing units. This is a major optimization to be able to handle continuous queries.

2.3.4 Distributed continuous query processing

Continuous queries are defined on a potentially infinite sequence of tuples called stream. They aim to process data as they arrive in the system and return results with low latency depending on execution environment. In addition to the query processing paradigm, it is important to distinguish different granularity of tuples processing.

MapReduce based systems   Among MapReduce-based systems for stream processing, two categories arise. First, there are multi-shot systems. They are extremely close from recurring query management systems because they are based on an aggregation of tuples in a buffer which is processed a single time. Some systems provide mechanisms to set the buffering time and the duration between two consecutive results through a sliding window [ZDL+12]. A sliding window is an abstract stream discretization defined by a size and a slide value. The size defines which tuples are considered for the computation and the slide defines the duration between two consecutive windows. Nevertheless, the use of sliding window implies redundant computation. Actually, when defining a sliding window of 30 seconds and a slide of 10 seconds, sub-windows of 10 seconds are shared by two or three windows and are computed as many time as required. A solution widely adopted by datastream management systems (DSMS) is to cache each sub-window result that is going to be reuse further[ZDL+12] [ASG+12] [LMD+11]. The main difference between multi-shot systems and recurring query management systems is clearly that tuples are not loaded from disk but received from an unpredictable stream source. This unpredictable aspect rises many issues on stream variability. Stream variations are from two types : input rate and value distribution. In both case, DSMS must face scaling issues. Actually, significant increase requires that the system is able to adapt resources and parameters for data buffering like dynamically adjust buffer size [ASG+12] or split buffering tasks on multiple nodes. At the opposite side, significant decrease can involve idle nodes which is expensive in a cloud environment as presented in [SHLD11]. Buffer issues can be avoided while using on-the-fly systems.

These systems, also based on MapReduce paradigm, are clearly different because tuples go through Map and Reduce phases as they arrive. Of course, it does not prevent to compute values on sliding windows. Instead concentrating tuples of a (sub) window in a buffer, punctuations are used. Punctuations are specific tuples injected by the DSMS into the stream to notice that all tuples have been received for a given (sub) window. On-the-fly systems face an issue directly linked to punctuation mechanism : synchronization. While a reduce node has to process result for a (sub) window, he has to wait that all map nodes have received punctuations from sources like illustrated in [BPFC12]. More, it is less easy to adapt dynamically sub-windows size than buffer size of multi-shot systems because it requires to inform all sources simultaneously.

Handling streams tuple by tuple is also the processing granularity adopted by box-and-arrow systems.

Box-and-arrow based systems   A box-and-arrow system takes as input one or many DAGs representing user defined queries and input streams. Streams may be shared among queries but they are processed in parallel as shown in [CBB+03]. As presented earlier, each DAG vertex's represents an operator and edges define operators order. Parallelism can rely physically on multicore architecture [CBB+03] or on a cluster of machines [CBB+03] [CCD+03]. Using workflows allows to define constraints like latency or result precision through a Quality-of-Service (QoS). Each workflow result has a score based on one or many parameters (computation time, precision, resources used...). A user sets maximal or minimal values on those parameters so it defines a QoS. It influences directly system outputs scores but also intermediate results scores, basically for each processing unit. Actually, inferred QoS [CBB+03] can be established from

a global one to set intermediate constraints which must be respected by each node to fulfil user defined constraints. More, QoS is often used for an optimal operators scheduling among processing units with regards to their respective costs [CBB⁺03] [CCD⁺03]. Finally, some strategies for latency decrease (sampling) or precision increase (redundancy) are used depending on intermediate scores. But, QoS is not the unique way to get a balance operators and data distribution. Auto-balanced systems can also be defined by economical contracts as described in [BBS04]. The main difference with QoS-based architecture is that each resource is linked with an economical value and some tasks to complete. If the resource is not able to fulfil all assigned tasks it can pay an other one to do some of his tasks. There is also an alternative for operators and data distribution presented in [XCTS14]. The main idea is to consider data transmission between processing units as an important vector of latency increase. So, the system dynamically tends to process, on less machines as possible, operators that exchange great amount of tuples.

Finally there are some hybrid systems based on workflows but handling data as MapReduce implementations [NRNK10] [SHLD11]. Those systems transform raw data into key/value pairs with some additional metadata essentially for scheduling and semantic enrichment. Then, those pairs are processed by dedicated nodes. For example, a node can be defined as a filter only for a specific semantic type and a subset of all possible keys [NRNK10]. The main benefit is to have a more expressive functions definition than Map/Reduce and to base computation on keys as MapReduce systems.

2.4   Querying sensor data.

Sensor data are measures produced by sensor devices. There exists a wide variety of sensors and sensor-like devices, from small scale sensors, like sensors embedded in today's smart phones, to very big measurement installations, like the Very large Telescope (VLT, by the European Southern Observatory).

Sensor data are inherently multidimensional, with 3 main common dimensions: Time, Location, Source. Each dimension can be described by a hierarchy of levels, like Seconds, Minutes, Hours, Day, Weeks for Time. As production of sensor data can last long periods of time and/or concern a large number of sensors, a measurement installation can produce large volume of data (Volume) and/or fast data streams of measures (Velocity). Measures themselves are often simple numerical values representing physical phenomena), but can also include multimedia data like pictures, sounds, videos or texts. And one measurement installation can also include different types of sensors, leading to a variety of data types and/or semantics (Variety).

Offline analysis of historical sensor data generally relies on Big Data technology to handle those large amount of data, and on Data Warehouse and OLAP technology to exploit their multidimensional properties. Real-time analysis of sensor data streams relies on (distributed) continuous query processing.

Sensor data processing systems, on historical data or data streams, are often organized in a centralized way (even with distributed processing), with raw data being gathered and then processed for a unique goal. Community aspects like sharing data in P2P and/or collaborative data processing are not common for sensor data.

2.5   Aggregative queries.

Initial work on decentralized aggregation protocol dates back to the early 2000's. The first papers deal with tree-based or otherwise structured peer-to-peer topologies. [GVB01] addresses the problem of computing aggregate functions of the data in large process groups by means of a decentralized hierarchical protocol. The authors explore two ways to build the hierarchy: one based on a leader-election approach, and the other based on a gossip protocol. Another early paper, [BGM03], proposes tree-based protocols to compute aggregate functions in a peer-to-peer network. Tree-based protocol also appear in many papers in the context of sensor networks, where aggregating data is important in order to save bandwidth and battery power. [CPS] proposes a hierarchical protocol designed to be resilient to malicious (byzantine) nodes. Astrolabe [VBV03] uses on-the-fly aggregation mechanisms to compute summarized data in the context of a large-scale gossip-based information-management system. Albeit gossip-based, the system still employs a hierarchical topology. [AM10] explores how to recover from failures in high-performance aggregation protocols deployed over a tree-based topology.

The performance of tree-based and structured protocols tends however to degrade in the presence of dynamic networks, resulting from mobility or churn. This have led researchers to explore completely

unstructured solutions based on gossip and random walk. Initially introduced as a family of algorithms for maintaining replicated database systems [DGH+87], gossip-based protocols have been applied in a wide range of situations such as data dissemination [EGH+03], overlay maintenance [JVG+07, VvS05], and even data-mining applications [BFG+13]. In a gossip protocol, each node periodically exchanges messages with randomly selected nodes in order to disseminate data, exchange membership information, or build distributed state.

Gossip-based aggreagation was initially proposed in [JM04, MJB04]. The papers present and evaluate a pairwise averaging approach in which pairs of nodes replace their current values with the average of their values. The authors show that this simple protocol converges effectively to the average in a large-scale network. Almost in parallel, another group of researchers [BGPS05, BGPS06] studied the same algorithm from a theoretical viewpoint and analyzed its convergence time in a variety of network conditions. Since then, several authors have expanded this work by examining how epidemic protocols fare with more complex forms of aggregation. For example, [MV06] proposes a protocol based on the idea of belief propagation, called consensus propagation. This protocol provides faster convergence than pairwise averaging in general graphs. [BHOT05] describes a generalization of the algorithm to compute arbitrary statistics in an incremental manner. Yet most of these newer algorithms have mostly been studied in theory and we are not aware of practical implementations in real systems. It will therefore be interesting to explore how they behave in the context of network architectures like the one we describe in Section 4.

Other related work includes [ADGK06], [HR08], and [AYSS09]. The first, [ADGK06] presents a sampling-based approach to distributed aggregation queries based on random walks. The second, [HR08] presents a gossip-based algorithm to estimate the distribution of values held by peers across a network. This allows a peer to evaluate its rank according to a particular metric, but the protocol may also be used to compute statistics over the estimated distribution. Finally, [AYSS09] presents a an algorithm for computing weighted averages through a gossiping protocol but in a wireless setting. Finally, some authors have started to address how to use decentralized protocols to compute aggregation functions in a privacy-preserving manner [BE13].

3   Overview of the SocioPlug system organization.

For an organization to be interesting, its contribution must be superior to its cost. In particular it has to avoid frequent and costly reorganizations. This means that it has to be based on elements themselves stable. This explain why rewriting technics have not been considered in the context of one-shot queries. Indeed, these queries have very short lifetime (in fact, an user requirement is for this lifetime to be as short as possible). This does not mean that queries are out of interest. Indexes, caches, and other structures playing an important role in the optimization process are build considering queries. Anyway, it is not realistic to build an organization directly over one-shot queries. This explain why Semantic Overlay Networks rely on other elements (as data).

The parameters change drastically considering the continuous queries. Indeed, these queries are stable elements (possibly even more constant than data) with a (very) long lifecycle. Consequently they become serious candidates to participate to a stable organization.

3.1   Overview.

In this section, we focus on Socioplug computation steps and their requirements. In our context, query execution physical support is composed of plugs with heterogeneous capacities (CPU, memory, bandwidth) located at distinct user places. We consider any continuous query computation as the combination of four endless phases :

- Data stream acquisition.

- Query execution on acquired data.

- Results dissemination among plugs involved in global query computation.

- Results storage for further local computations (considering plugs which perform a computation) or for other communities which could benefit from these results.

3.1.1   Tracker.

A tracker functionality maintains information reflecting the current state of the system and can be queried about them. It is the main entry point of the system: an incoming participant may ask for the list of communities that already compute a query that is equivalent to the query he/she is interested in, or for the whole list of queries for which at least a community currently exists. It also maintains a history of the system, which can be queried too and helps provide statistics about the system.

For each query, the tracker functionality should provide its description (syntax and semantics), the list of sources it involves, and the list of communities that compute it with the corresponding result throughput. For each community, it should provide an entry point (or a list of its participants), so that a participant can ask to join it.

The tracker functionality can further be queried about basic general information such as the number of participants, the number of sources, the number communities... More complex analysis queries can also be asked, like asking for the ten attributes that are the more present in running queries, the ten most used sources last month, all the running queries that use a particular set of sources...

In the system, this tracker functionality may be implemented as a centralized process and/or as a distributed process among communities or participants.

3.1.2   Data acquisition.

Instead letting tuples run into query computation layer as they arrive from sources, a data acquisition layer handles them in order to ensure that query computation layer is able to process streams. First, this layer provides stream source connectors necessary to receive input streams. As [w7w] or [ZDL$^+$12], these connectors can be be generic to process raw data or specific like Twitter connector to receive data with associated metadata. Second, data acquisition layer is in charge of supporting data stream variations and fit resources accordingly in order to prevent overload during next data treatments. Input streams rate variations can cause overload when rate increase or almost idle nodes while rate is low. Acquisition layer must then adjust active acquisition node pool to consume only necessary resources. Finally, most applications require to analyse simultaneously multiple heterogeneous streams. It rises two important problems : on one hand, data must be integrated into a unique model that can be used by query computation layer. On the other hand, acquisition layer must have a up-to-date knowledge base about sources (metadata and semantic information). This base is used to update integration models. Resources adaptation issues are considered in existing solutions as MAPE loop [SHLD11].

3.1.3   Query computation within a community.

Query computation layer receives as inputs a user-defined query and data coming from acquisition layer. Firstly, this layer has to get connect to appropriate acquisition nodes. It may be an issue if many nodes provides redundant data. Actually, this redundancy can be expensive to delete and lead to wrong results while computing queries at the same time. Then, even if acquisition layer is already in charge to regulate stream variation it does not mean that load can not increase or decrease on some computation nodes. It only means that an input stream rate variation will not impact immediately query computation layer but will involve greater or smaller amount of data to process later. The delay must be used to adapt computation resources to a different amount of data to process. It leads to plan a new pool configuration which potentially requires more or less active computation nodes. This new configuration may involve tasks redistribution and data transmission information to route data efficiently according to associate operators. Considering that the pool is composed of heterogeneous nodes, each resources must be described in order to benefit as much as possible from each active resource potential (CPU, memory size, bandwidth...). Those aspects are considered partially in existing solutions [BPFC12] [ZDL$^+$12] [w7w] [CCD$^+$03].

3.1.4   Result dissemination.

Result dissemination layer aims at distributing final result on each node belonging to a community. In order to share result as fast as possible it is essential to find the less expensive distribution roadmap in terms of bandwidth. As query computation layer, resources are considered heterogeneous so it is necessary to consider resources properties, especially bandwidth. Those aspects are considered in existing solutions as [KvS07].

3.1.5   Results storage.

Results storage layer is important inside and outside a community. It requires to identify which results will matter later for the community which compute it or for an other.Actually, intermediate results can be stored to avoid redundant computation in a community like sub-windows computation shared by many sliding windows. In this case, it is important to consider computation which will use intermediate results in order to store them on the same node. It avoids useless transmission latency. For intermediate and final results, it is necessary to keep indexes and metadata on results in order to identify them efficiently. It the case where results can not be stored on nodes which will use them, additional information are necessary to access the appropriate storage node. Those aspects are considered in existing solutions [BPFC12] [LZRE14].

3.1.6   Transversal/global requirements

For all layers, there are some global requirements that must be considered. First, strategies for fail-over management are required. Actually, a DSPS must recover a failed node without losing data or at least be aware of lost data to transmit this information to end-users. Then, a global pool management strategy is required to have the appropriate parallelism level. Two vision are opposed while distributing treatments : on one hand, treatments are spread on as many nodes as possible. On the other hand, each node is used as much as possible. A system can switch between these two strategies to fit resources to computation.

3.2   Problem definition and general approach.

3.2.1   Problem statement.

In this section, the problem is formally stated on the basis of query rewriting. To keep general, we do not assume a particular language to express streams and queries. We only assume any stream can be labelled by a query that represents its content.

Query rewriting.   For the sake of generality, we rely on the notion of query rewriting (as compared for example to the notion of containment as used in some pub/sub systems). Describing rewriting technics is far beyond the scope of this report. We just point out the few definitions and notations that will be used.

Two queries, q_1 and q_2 are equivalent when they provide the same results, regardless of the content of the data streams. This is noted $q_1 \equiv q_2$.

In our context, rewriting some query q using queries (we could say views) $q_1, q_2, \ldots, q_n$ means there exists a query q' expressed over these q_i, equivalent to q. This is noted $q' = (q \longleftarrow \{q_1, q_2, \ldots, q_n\})$, or simply $q \longleftarrow \{q_1, q_2, \ldots, q_n\}$, when q' is of no need.

Notice that from $q_1 \longleftarrow \{q_2\}$ and $q_2 \longleftarrow \{q_1\}$ one can conclude that $q_1 \equiv q_2$.

Definition 3.1 (Rewriting schema).   *A rewriting schema of a query q_r, is a graph $\langle q_r, Q, E \rangle$ such that*

- *Q is a set of queries, with $q_r \in Q$,*

- *E is a set of hyper-arcs such that*
  $\forall q \in Q$, and $\forall \{q_1, q_2 \ldots q_n\} \in Q^n$ if $(\{q_1, q_2 \ldots q_n\}, q) \in E$ then there exists a rewriting $q \longleftarrow \{q_1, q_2, \ldots, q_n\}$,

- *the in-degree of each node is at most 1,*

- *q_r is the only node with an out-degree equal to zero,*
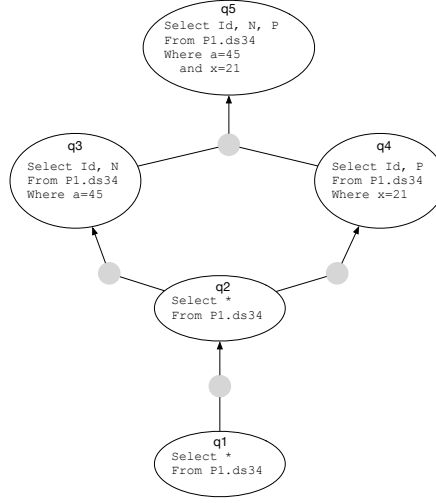
- *the graph is cycle free and totally connected.*

Figure 3: Rewriting Schema.

Figure 3 illustrates a rewriting schema where *q*2 is rewritten from *q*1 using identity (`select * from q1`) ; *q*3 and *q*4, is rewritten using *q*2 adding a selection and a projection (`select Id, P from q2 where x=21`) ; *q*5 is rewritten as the intersection of *q*3 and *q*4. There may be many possible rewritings between queries, but as the configuration graph used in [KKM13], the rewriting schema contains at most one possible rewriting for each query and avoids cycles.

System definition.   The role of a distributed system is to provide *users* with the information they require, *i.e.* to compute answers to their queries. It is composed of software *participants* which originate some streams, submit users' queries, and bring to the system some resources like memory, storage and computational capacities.

Definition 3.2 (System). *A system is a labelled graph $\langle P, PA \rangle$ such that:*

- *P is a set of participants. For each participant $p \in P$,*

    - *p.sources is the set of queries representing the data streams it originates in the system,*

    - *p.queries is the set of queries it submits to the system*

- *PA is a set triples $\langle p_i, p_j, Q_{i,j} \rangle$ such that*

    - *$(p_i, p_j) \in P^2$*

    - *$Q_{i,j}$ is a non empty finite set of queries representing the data streams p_j gets from p_i.*

In addition, we define the following sets for any participant *p*:

- *out(p)* is the set of data streams *p* communicates to some other participant.

- *in(p)* is the set of data streams that *p* gets from other participants.

- *comp(p)* = $((p.queries \cup out(p)) \setminus in(p))$ is the set of queries a participant has to compute.

Figure 4 illustrates a distributed system where *p*1 is the unique participant that originates a stream (represented by *q*1). *p*2 submits queries *q*2, *q*4 and *q*7 to the system. It gets *q*1 form *p*1 and *q*7 from *p*4. It computes *q*2, *q*3, *q*4 and *q*5. It serves *q*1 to *p*4 ; *q*2 to *p*3 and *p*5 ; *q*3 to *p*4, *p*6 and *p*7 ; *q*4 to *p*5 and *p*6 ; *q*5 to *p*4 and *p*7. And so on for other participants. All this has to be coherent with possible rewritings.
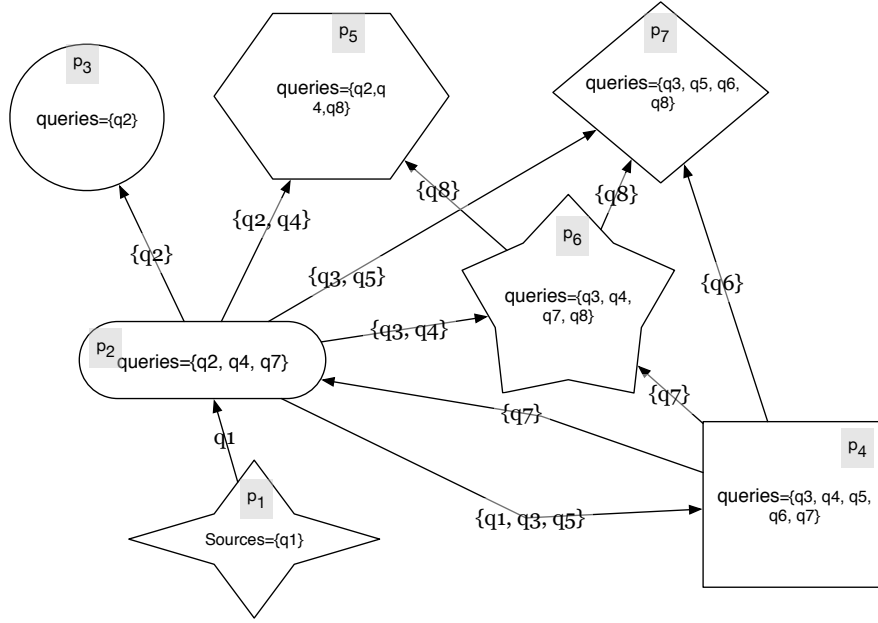
Figure 4: System example with seven participants.

Definition 3.3 (Implementation of a rewriting schema). *A system $\langle P, PA \rangle$ implements a rewriting schema $\langle q_r, Q_r, E_r \rangle$ for a query q issued by a participant p if and only if*

- *$q_r = q$ (syntactic equality).*

- *there is a mapping m from Q_r to P such that*

  - *$m(q_r) = p$*
  - *$\forall (\{q_1, q_2 \ldots q_n\}, q_g) \in E_r, \ \forall q_i \in \{q_1, q_2 \ldots q_n\}, \ (m(q_i) = m(q_g)) \ or \ (\langle m(q_i), m(q_g), Q_{i,g} \rangle \in PA$ and $q_i \in Q_{i,g})$*
  - *$\forall q_i \in Q_r$, if q_i is a leaf then $q_i \in m(q_i).sources$*

The objective is to provide each participant with data streams to compute its issued queries. In our approach, these data streams may be different from those expressed in the original queries. Indeed, query rewriting technics provide such solutions.

The global organization has to be globally coherent, and for example, has to avoid cycles. Indeed, even if a query q can be rewritten based on another query q' is not a sufficient evidence to conclude it is possible to proceed that way in the system. For example, the query q =select * from DS42 where x=21 or y=72 can be rewritten into select * from q' assuming that q' =select * from DS42 where y=72 or x=21. This is an interesting writing for q since it takes advantage of the job already done by q'. However, before to build the system on this rewriting, it is necessary to verify that the query q' is not itself rewritten using q, just for the same reasons. If the system makes use of these two rewriting at the same time, these two queries will wait for results one from the other. This is not the only way to introduce a cycle. Things may be less direct and go through hyper-arcs. For example, the query q can also be rewritten as the union of the two queries q_1 =select * from D21 where x=21 and q_1 =select * from D21 where y=72. If one of these queries is rewritten based on q (for example select * from q where x=21 could be an interesting rewriting for q_1, especially if q provides less data that DS42) also introduces a cycle. Cycles are important to avoid, because no matter on how they are produced, by direct writing or by less direct fashions, a cycle always makes involved queries to wait for results one from the others such that

no one would ever produce nor get any result. Confronting a system with rewriting tree is a way to verify that it does not embed any cycle.
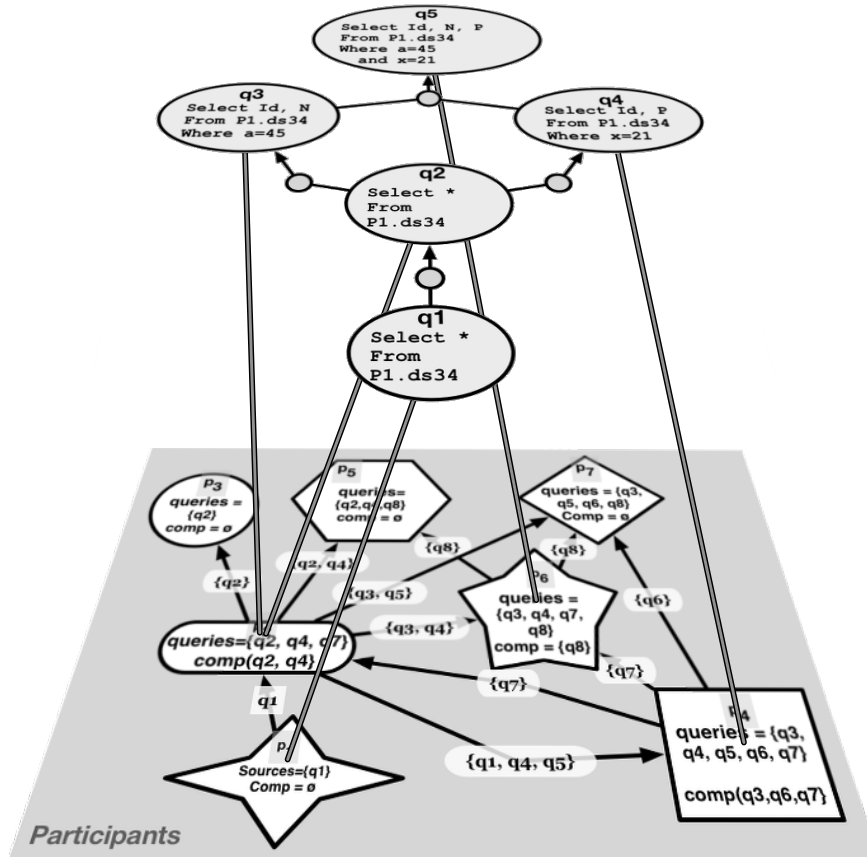

Figure 5: Example of a system implementing a rewriting schema.

Definition 3.4 (Well founded system). *A system is* well founded *if and only if it implements a rewriting tree for any issued query q.*

Beyond this theoretical definition, in order to further define the problem it is necessary to pay attention to physical constraints. The participants have limited computation and network capacities. Overload of each element has to be avoided. Moreover, the solution should consume as few resources as possible considering not only its operating cost, but also the setting up and maintenance costs. In addition we assume that each participant has a resource management policy which specifies to which uses the resources that it provides are dedicated. Thus the solution has to comply with these policies. Last but not least, in many applications, latency is of prime importance for users' welfare. In our case, this point has to be considered too, looking for a latency as minimal as possible.

Problem statement   The problem is to find a *well founded system*, which is 1 - coherent with participants capacities, 2 - compliant with their resource management policies, 3 - as efficient as possible with respect to resources consumption (computing, network, energy), and 4 - providing results to participants with a latency as small as possible.

Unsurprisingly, finding a solution that minimizes simultaneously costs and latency while satisfying other constraints may be out of reach or very difficult.

3.2.2   General approach: Communities.

In its current expression, the problem seams rather complex. In this section, we describe our way to rephrase it into several simpler steps. Our approach is first to draw the system general structure, based on query equivalence and query rewriting, such that the remaining problems could be solved locally without questioning again the general structure.

Abstracting participants and queries to computing units and communities.   First, it is not difficult to understand that a software participant is a composite module, mainly due to the fact it is associated to many different queries. For ease of structuration, we introduce the notion of *computing unit* or simply *unit*. Such a unit is part of a participant, but in contrast, it is focused on one and only one query. Intuitively, a unit is created each time the participant p (or its owner) explicitly declares some interest in a query q. From a technical point of view, it corresponds to resources that a participant p allocates to the management of a query q.

Definition 3.5 (Computing Unit)**.** *A computing unit is a couple $u = \langle q, p \rangle$, where p is a participant of the system, $q \in p.queries \cup p.sources \cup comp(p)$.*

Notice that this notion does not conflict with, nor limits in any way, the possibilities of optimization of a participant's activities. For example, if a participant is in charge of the computation of many queries it can implement optimization solutions like those proposed in [TATV11, CATV11], regardless of the use of the 'computing unit' concept in the formal analysis and system organization. Nevertheless, it's up to each participant to allocate resources to its computing units (computation, memory, network...).

It is an evidence that the number of units is greater than the number of participants. Clearly, expected advantage is not to reduce the graph size but to make easier the grouping with respect to common interest, i.e. issued queries. Indeed, gathering computing units associated to equivalent queries in communities is expected to bring a number of benefits.

First, beyond the clustering of units, focusing on communities should allow to reduce the number of node (compared to units, and potentially compared to participants) such that the increase of the graph size should be limited. Indeed, in presence of popular queries, the number of communities can be dramatically lower that the number of units, such that focusing on communities will give a synthetic semantic description of the system.

Second, within a community, strategies, protocols and algorithms can be developed to share resources and to manage the treatment of the query everyone is interested in (acquiring data streams, computing, diffusing results, storing). This is an opportunity to look at ways to compute complex queries that are out of reach of a single participant. Indeed, the more popular a query is, the more numerous the units are in the related community, the more resources are available.

Definition 3.6 (Community)**.** *A community C is a nuplet $C = \langle q, U \rangle$ where*

- *q is a query*

- *U is a set of computing units such that $\forall u \in U$, $u.q \equiv q$.*

Notice that for many different reasons, such as language complexity, all equivalences are not provable in a reasonable time. This explains why we do not require all units with equivalent queries to be part of the same community. However, the most important point is warranted: all units of a community share common interest in the same results.
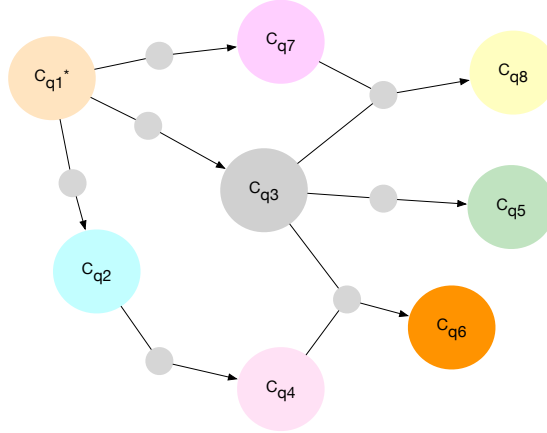
We call *source-communities* the communities which contain a source unit together with units that consume the whole data stream it produces. In such communities, a source with poor resources may rely on other units to help it to transmit its data stream.

By gathering units, and therefore participants, according to the queries they share, communities can be considered as some kind of semantic overlay over continuous queries. As far as queries play a central role in community definition, this overlay can be structured taking advantage of rewriting technic. We are back here with the intuition that has been used to define the problem. The new point is the use of communities which are focused on one and only one query. This single point helps to clarify and to simplify the introduction of rewriting rules in the picture.

Definition 3.7 (Graph of Communities). *A Graph of Communities is an hyper-graph $\phi = \langle C, RR \rangle$, such that*

- *C is a finite set of communities where $\forall (c_i, c_j) \in C^2$, if $i \neq j$ then $c_i \cap c_j = \varnothing$,*

- *RR is a set of hyper-arcs such that:*

  - *if $(\{C_1, C_2 \dots, C_n\}, C) \in RR$ then there exists a rewriting $C.q \longleftarrow \{C_1.q, C_2.q, \dots, C_n.q\}$,*

  - *the in-degree of any source-community is equal to zero ; the in-degree of any other community is equal to one.*

  - *the graph is cycle free.*

Source-communities are labelled with a star.

Figure 6: Graph of communities

The graph of community is a structure with properties related to rewriting which are equivalent to the one presented section 3.2.1 on the system.

Theorem 3.1 (Graph of communities and rewriting schema). *Considering a graph of community $\phi = \langle C, RR \rangle$, for each community C, the graph describes and implements a rewriting schema.*

Proof. Let us consider one community C of $\phi = \langle C, RR \rangle$. The proof is split up into two parts.

- ϕ describes a rewriting schema

  Let us consider the subgraph $\phi_C = \langle C_C, RR_C \rangle$ obtained as follows.

  - initialization
    * $C_C^0 = \{C\}$
    * $RR_C^0 = \varnothing$
  - construction
    * $C_C^{i+1} = C_C^i \cup \{C' : \exists (\{C_1, C_2 \dots, C_n\}, C_r) \in RR \text{ and } C_r \in C_C^i \text{ and } C' \in \{C_1, C_2 \dots, C_n,\}\}$
    * $RR_C^{i+1} = RR_C^i \cup \{(\{C_1, C_2 \dots, C_n\}, C_r) : \{C_1, C_2 \dots, C_n\}, C_r) \in RR \text{ and } C_r \in C_C^i$
  - finalization.
    Let k be the smaller integer such that $C_C^k = C_C^{k+1}$ and $RR_C^k = RR_C^{k+1}$.
    * $C_C = C_C^k$
    * $RR_C = RR_C^k$

This integer always exists due to the fact that the community graph is finite.

Since, ϕ_C is a subgraph of the community graph,

- to each node of C_C there is an associated query, set of which can be noted Q.
- to each hyper-arc of RR_C there is an corresponding rewriting rule, set of which can be noted E.
- the in degree of each node is at most 1.
- C is the only node with an in degree equals to 0.
- the graph is cycle free and totally connected.

In other terms, it is a rewriting schema.

- ϕ implements a rewriting schema

  Showing that the system implements a rewriting schema is trivial, there is no need of mapping: the rewriting schema is part of the community graph by construction. The only point to verify is that leaves are sources.

  Since in the community graph the source-communities have an in-degree of zero, if they appear in the sub-graph, they can only be leafs.

  Considering that the in-degree of other communities is equal to one, the community graph is finite and cycle free, sources-communities are the leaves and the only ones to be leaves in the community graph.

  As a conclusion, the rewriting schema leaves are source-communities.

This proof can be conducted for any community. So we can conclude that the community graph is well founded with respect to rewriting rules. □

Considering our use of rewriting, one can think of directly organizing units without involving communities. Indeed, at first glance, problems may look similar, but some differences have to be pointed out.

First, in presence of popular queries, the number of communities can be dramatically lower that the number of units. Therefore the concept of community scales down the number of nodes to be considered, and a smaller problem is easier to solve.

Second, queries regrouped within a community are by definition equivalent queries. Thinking in terms of communities pushes the rewriting rules linking them to each others out of the picture. This a double good news. Indeed, finding rewritings is usually a difficult task and avoiding to find some of them is already a good point. Furthermore, since these are equivalent queries, rewriting usually introduces many cycles which have to be tracked and handled. Both of these operations are complex, and it is a very good point to avoid them.

To conclude about graph considerations, despite their theoretical similarity, to obtain the graph of communities is easier than to compute directly the graph of units.

A third point to consider is the role of communities. Indeed, introducing communities is an opportunity to make units collaborate to compute complex queries which would be out of reach of a single participant. The more popular a query is, the more resources are available. More generally, thanks to the community base approach, the issues of query computation and result dissemination are no longer global problems. Instead, each community has to solve them locally and independently of other communities. The advantages are a simpler local resolution of smaller problems, the autonomy of participants and the scalability of the approach.

To summarize, based on the concept of graph of communities, the problem organizing a system can be rephrased in three steps: *(i)* defining the graph of communities, taking advantage of rewriting queries, then *(ii)* specifying resource allocation and the stream exchanges between communities and finally *(iii)* locally organizing each community. Then, the system itself can be deduced rather simply.

(a) Publish/Subscribe approach.

(b) Creating a new artefact on the bottom community.

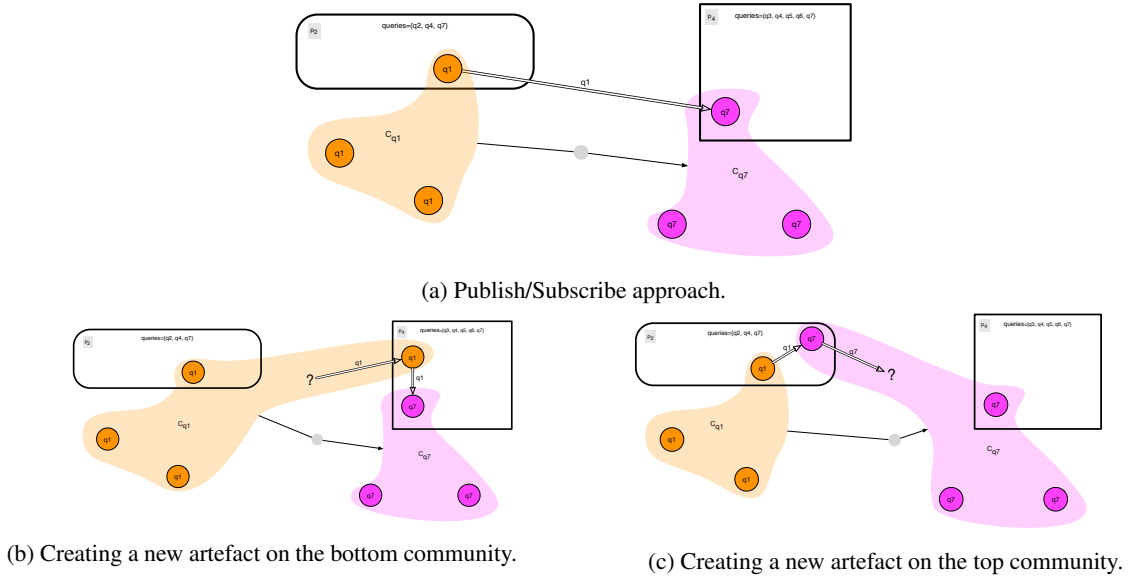(c) Creating a new artefact on the top community.

Figure 7: Carrying a data stream from one community to another.

3.2.3 Data streams exchanges between communities.

Based on the general picture, a first question to answer is how does a community get the data stream produced by another one. This problem occurs each time a community is defined with a rewriting rule involving other communities. For example, $(\{C_1, C_2 \ldots, C_n, \}, C) \in RR$ means that community C needs to get the data streams produced by each community C_i.

As illustrated in figure 7a, a straightforward solution is that a unit of C (p_4's unit in community C_{q7}) directly contacts a unit of C_i (p_2's unit in community C_{q1}) for subscription to the needed data stream ($q1$ in figure 7a).

This approach consumes a communication resource of community C_i. When the number of subscriptions exceeds the capacity of the community C_i some will not be able to access this resource, resulting in a famine situation. Furthermore this raises a question about the use of participants' resources. Indeed, a participant joins and brings resources to a community just to obtain results. It is natural that it participates in the management of the query from the computation to the dissemination of results within the community. It is more questionable that it agrees to provide resources for dissemination of results to the entire system. Famine problem combined with interrogations about motivation justify the need to explore other ways.

To avoid motivation interrogation, we limit the dissemination to the community only. To avoid the famine problem, we ask community C to compensate the resources it consumes (i.e. to redistribute data stream of the community at least one time.). These two points argue in the same direction: if a participant wants to obtain results from a community, it should consider to be part of it and to bring it at least as much ressources as it consumes. This leads some participants related to community C to create new units, one for each C_i (see figure 7b). These new units play a role of carrier, and inter-communities data stream exchanges are managed inside participants. Separately from all other matters, these units are members of a community, and as any unit, they can participate to the community tasks (acquiring data, computing, result dissemination and storage) up to their capacities.

Lastly, we can notice that famine is not always an imminent risk. In some cases, a participant related to C_i may have enough resources to support the communication cost with community C as well as the computation of the query of C. It may even be less expensive for it than broadcasting the C_i data stream (for example, due to a high selectivity of an easy to compute operator). In such a situation, a participant related to community C_i may have interest to ensure by himself the communication with C. To do so, it can create a new unit which is introduced into community C (see figure 7c). Its role is to compute the query of

community *C* and to communicate the resulting data stream to some other *C*'s units.

Several reasons can lead a participant to positively answer to the solicitation of an outside community. First, as explained above, resources consumption can be an objective reason. Second, a participant may have some interest in the others getting quick results (as for example the participant providing a data stream). Third, altruism is also a possible motivation. Fourth, in case of general interest (for exemple network or energy consumption), some rewards can be introduced by the system to encourage such behaviour.

In summary, we retain two ways to enable a community to get data streams from others. Both are based on the creation of new units which become members of a community. Choosing one or the other solution is a problem which has to be solved considering specific situations, so this decision is left open. Whatever the result of this negotiation is, the newly introduced units are not differentiated from the others. Each one participates up to its capacity to the different tasks its community is in charge of.

3.2.4 Organizing a community.

The global organization movements correspond to communities arrival, departure or moving from one rewriting to another. All these movements are implemented by movements of units (arrival, departure) in communities. Thus, from a community point of view, there are only movements of units. This means that each community can manage autonomously its local organization considering units members regardless any other consideration. So, communities can manage independently from each others. This schema is adapted to a distributed and parallel approach.

At a local level, each community has to be organized to meet its basic needs: 1 - to obtain necessary data streams ; 2 - to compute the result ; 3 - to broadcast the result to all the members of the community ; and 4 - to store the results.

As already explained, gathering units in communities gives the opportunity to organize and manage problems which are out of reach of a single participant. However, as long as it is possible, it is easier for both data stream acquisition and query computation, to be processed by a single unit. In a first stage, we make the hypothesis that issued queries can be handled that way. Distributed solutions will be introduced in a second stage to handle more complex queries.

Disseminating the result to the complete community is a problem which, by nature involves all units of the community. It is interesting to note that under the hypothesis that each computing unit has at least enough resources to communicate the result of its *q* to another computing unit, this problem will always have a solution. The challenge will be to optimize the latence for participants.

The problem of storage is optional in the sense that the system can work without it. As the previous ones, this problem can be approached by a centralized approach (involving only one unit) or a distributed approach (involving up to the community in its whole).

All these points are matter of distributed algorithms and protocols. They will be addressed in the next deliverable.

3.2.5 Getting the whole system definition.

Once *(i)* the graph of communities is known, *(ii)* all the units are placed in their communities, including those related to data stream exchange between communities, and *(iii)* each community is internally organized with respect to computation and data stream dissemination, the corresponding system can easily be defined. Indeed, *(i)* what a participant has to compute, *(ii)* which other participants it gets data streams from and *(iii)* which ones it disseminates data streams to, are directly deduced considering its units. Of course, a participant can optimize its computation and dissemination, but this is an internal problem that does not affect the system definition. This step is simple enough to avoid formal definition.

3.3 System adaptation to different capacities and strategies of participants.

An interesting property of our proposal is its ability to adapt to different situations and participants' strategies. To illustrate this point, we consider a simple situation involving several subscribers and a single producer *p*1 that provides a data stream represented by query *q*1. Participants' issued queries are summarized in table 1. We assume that the rewriting phase results in the graph of communities of figure 6.

Table 1: Issued queries in the system

	$q2$	$q3$	$q4$	$q5$	$q6$	$q7$	$q8$
$p2$	X		X			X	
$p3$	X						
$p4$		X	X	X	X	X	
$p5$	X		X				X
$p6$		X	X			X	X
$p7$		X		X	X		X

(a) Using units and participants to exchange data streams between communities.

(b) Participant point of view after organizing diffusion within communities.
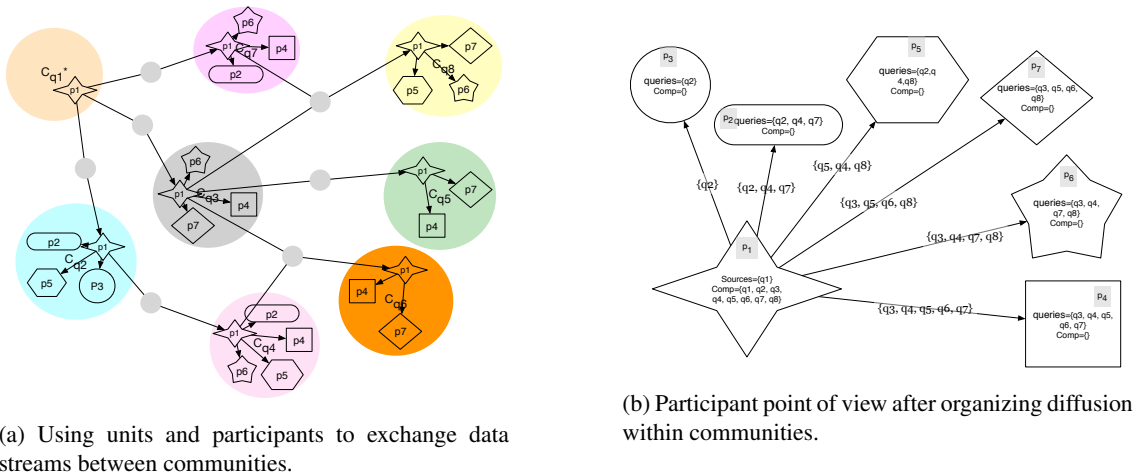
Figure 8: Powerful data stream producer that accepts to compute for others.

In the following we consider three scenarios showing how differences on participants' ressources and strategies may lead to different systems. In our figures, a participant is represented by a specific form (e.g. circle, star...) which is also used for its units with a smaller size.

In the scenario corresponding to figure 8, the data stream producer is powerful enough to compute all queries and to disseminate results. By altruism or by interest, this producer always answers positively to other communities demands. In this scenario, other communities always ask for help, such that $p1$ creates units on each community leading to sub-figure 8a. As far as $p1$ computes all queries and sends results, other participants get their results directly from the data producer without any effort. Sub-figure 8b describes the system using the participants point of view. It depicts a classical publish/subscribe system where the data stream producer also plays the role of the broker.

The scenario of figure 9 starts from the same graph of communities presented figure 6. The differences are the participants' characteristics. The data stream provider can not compute anything and can deal with only out going connexions[†]. All participants are self-serving and so do not accept to work on queries which are not involved in the computation of their issued queries. Consequently, they systematically give a negative answer to other communities demands. This leads to the situation depicted on figure 9a. Focusing on participants point of view, figure 9b, shows a peer-to-peer oriented system, which is very different from the previous one even if obtained from the same graph of communities (figure 6). As previously each query is computed only one time, but here it is a participant issuing it which is in charge.

In the last scenario of figure10 the producer can send its data stream to many units but it cannot compute anything. Other participants are powerful and altruist. As shown on figure 10b participants $p2$ and $p6$ bear all the queries computation and most of the diffusion. The situation could even be more centralized

---

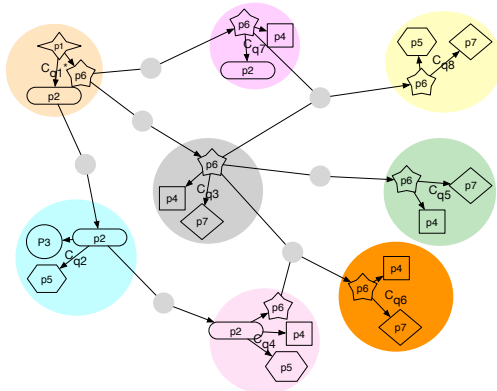[†] Such limits could as well be due to the participant's strategy.
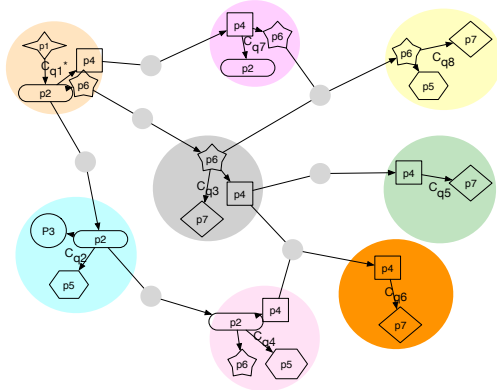
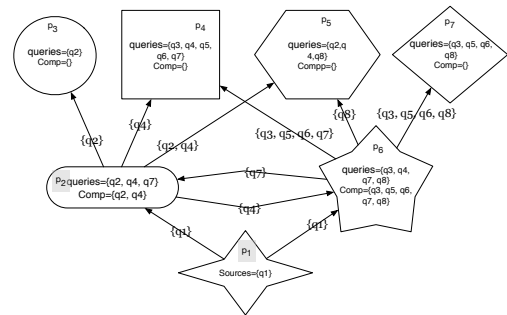(a) Using units and participants to exchange data streams between communities.
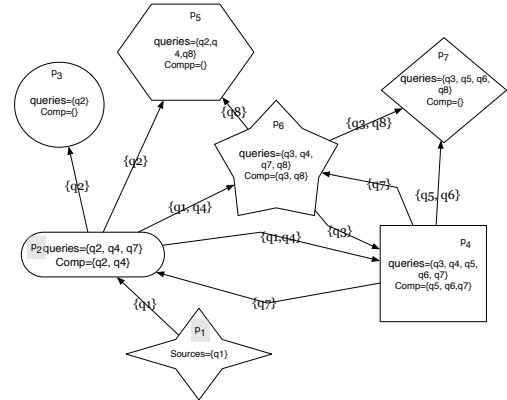
(b) Participant point of view after organizing diffusion within communities.

Figure 9: Powerless data stream producer with no altruist participants.

(a) Using units and participants to exchange data streams between communities.

(b) Participant point of view after organizing diffusion within communities.

Figure 10: A data stream producer with no computation capacity with altruist consumers.

with only one participant, for example $p6$, computing all queries and sending results to every one. This corresponds to systems with data stream providers, brokers and subscribers.

Based on fairly simple concepts such as graph of communities and the creation of units to provide data streams to communities, the formalism captures systems previously considered as very different. As a consequence, our approach is very flexible providing a continuum from centralized to fully decentralized solutions. The aim of the implementation will be to choose a solution according to the participants' capacities and strategies and to dynamically adapt to changes.

4 Aggregative queries.

Responding to aggregation queries in a decentralized system relies on a number of sub steps. Given a query, we first need to identify the potential data sources whose data may contribute to the query result. Second, we need to disseminate the query to these sources; and finally we need to aggregate and collect the results.

Our architecture for decentralized aggregative queries relies on a stack of epidemic protocols. Successfully used in a wide range of applications, epidemic protocols have served purposes ranging from data dissemination to decentralized computation.

Random Peer Sampling   The lower layer of the architecture consists of a so-called random-peer-sampling (RPS) protocol [JVG⁺07]. The RPS ensures connectivity by building and maintaining a continuously changing topology. This provides each plug with a dynamic set of neighbors that respresent a random sample of the network.

At each node *n*, the RPS protocol maintains a *view* data structure containing references to other nodes: the RPS neighbors. Each entry in the view is associated with a node and contains, at a minimum, *(i)* its IP address, *(ii)* its node ID, as well as *(iii)* a timestamp specifying when the information in the entry was generated by the associated node. Periodically, the protocol selects the entry in its view with the oldest timestamp [JVG⁺07] and sends it a message containing an entry for itself together with half of its view. The receiving node renews its view by keeping a random sample of the union of its own view and the received one.

Clustering   A clustering protocol operates in a similar manner to the RPS protocol described above. Each node also maintains a data structure called view and periodically exchanges the its content with other nodes. However, unlike the RPS protocol, a clustering protocol does not select nodes randomly, but according to the data hosted by the node itself. Nodes maintain a view data structure like in the case of the RPS, but each entry in the view also contains a digest of the information available at the node. Such a compact digest (for example in the form of the bloom filter) allows nodes to compute a similarity value with each of the nodes in their views. When a node receives the content of the view of another node, it does not choose randomly, but it selects the references that correspond to nodes that exhibit the highest similarity with the node's own profile.

Clustring protocols such as this have been used for a variety of applications including query expansion [], web search [], or recommendation []. In the context of this project, they will make it possible to identify communities of peers over which to compute aggregate functions. To achieve this, we will combine them with a third group of protocols: gossip-based aggregation.

Aggregation and Query Dissemination   The third type of gossip-protocol we will employ in our architecture deals with information aggregation. Originally introduced in [JM04], these protocols make it possible to compute aggregate functions such as the mean, a sum, or a variance by means of pairwise message exchanges between nodes. In our context we will augment these protocols with the ability to perform computations on private data.

Let $\P$ be a set of peers and $\mathcal{V}$ a universe of possible *values*. Let $v : \P \to \mathcal{V}$ be a function associating a peer, $p \in \P$ with a value $v \in \mathcal{V}$. At this level of abstraction, we can ignore the exact nature of v and of the set $\mathcal{V}$. A value may anything: a number, but also a collection of numbers, or a set of attribute value pairs. Given a set of peers $G \in 2^P$, we denote the multiset of values associated with the peers in G as (G).

Let $\mathcal{V}^{\mathcal{M}}$ be the set of all multisets over $\mathcal{V}$. Then an aggregation function is a function $\gamma : \mathcal{V}^{\mathcal{M}} \to V$ associates multiset of values from $\mathcal{V}$ with an aggregate value. Based on this, we model an aggregation protocol as a function ρ over a set of peers and their values. Specifically, $\rho : 2^P \to V$ associates a group of peers $G \in 2^P$ with an aggregate value v_G such that $v_G = \gamma(v(G))$.

Preserving Privacy in aggreagate computation   .

Our goal consists in identifying scalable decentralized aggregation protocols that limit the amount of information leaked to other peers during the computation of the aggregate information. To achieve this, we will combine the above protocol architecture with techiniques such as homomorphic encryption or Secure Multiparty Computation. Specifically, we plan to use ecrypted computation to secure each information exchange occurring between nodes in the course of an aggreagation protocol. This will make it possible to minimize the amount of information leaked to other peers while computing aggregate information.

5   Conclusion.

New technologic solutions enabling a user to deploy low-cost private resources of storage and computing with high availability are now onto the market. In this report we focus on solutions to the data querying problem relying on such resources with an approach based on community sharing which is complementary

to centralized solutions.

By combining the proposed definition of community with participant-based connections between communities, the *QTor* system turns out to be generic and flexible. Indeed, it adapts to different situations. For example, if a data service has low capacity, *QTor* minimizes the load on it down to one connexion. On the opposite, if a participant has huge capacities, and if its user agrees, its direct environment can take advantage of these resources very simply.

QTor enables a community to reuse the results obtained from others communities. So even if a standalone query is too complex for a participant, it may become tractable taking advantage of already computed results. Naturally, this possibility has to be supplemented by computing a continuous query in a collaborative way within a community. We will pay more attention to this part in future work.

The aggregate queries respectful of privacy are a special case of data querying, but this case is particularly important. Indeed, private cloud solutions can locally store data obtained from sensors and connected objects such that users can exploit their own data locally. This is of main interest: it improves confidentiality and reduces energy consumption due to the transportation of data, queries, answers, etc. Even if users currently allow their data to be centralized at suppliers', even if they are actually likely to reveal their queries to these suppliers, and even if it is conceivable that they allow public access to some of their data (ex. weather station), the hypothesis that a user provides full public access to her data is not realistic. Yet the analysis of these data presents clear interest. In this report, we have outlined how to implement aggregative queries in a community spirit to enable private data analysis via aggregative queries, without revealing those private data.

The next step will be the implementation of such systems. This requires the study of algorithms and protocols. Although the definitions proposed here lay the outlines, there are still open questions. For example, regarding the QTor organization a choice has to be made between two operational solutions: to regularly achieve an overall system optimization, or to incrementally build the system making it to evolve to adapt to changes. An organization produced by the first approach may be optimal, or at least very efficient with respect to queries computations, however it can be very costly in terms of system reorganization. Such issues are left open and will be under focus in the next step.

6 Bibliographie.

[ABW06]   Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

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

[AD98]   Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 254–263, New York, NY, USA, 1998. ACM.

[ADGK06]   B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki. Approximating Aggregation Queries in Peer-to-Peer Networks. *22nd International Conference on Data Engineering (ICDE'06)*, pages 42–42, 2006.

[AGD+06]   E. Anceaume, M. Gradinariu, A.K. Datta, G. Simon, and A. Virgillito. A semantic overlay for self- peer-to-peer publish/subscribe. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 22–22, 2006.

[AH01]   Karl Aberer and Manfred Hauswirth. Peer-to-peer information systems: Concepts and models, state-of-the-art, and future systems. *SIGSOFT Softw. Eng. Notes*, 26(5):326–327, September 2001.

[AH02]   Karl Aberer and Manfred Hauswirth. An overview of peer-to-peer information systems. In *WDAS*, volume 14, pages 171–188, 2002.

[AM10]   Dorian C. Arnold and Barton P. Miller. Scalable failure recovery for high-performance data aggregation. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010.

[And04]   David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[ASG+12]   A.M. Aly, A. Sallam, B.M. Gnanasekaran, L. Nguyen-Dinh, W.G. Aref, M. Ouzzani, and A. Ghafoor. M3: Stream processing on main-memory mapreduce. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1253–1256, April 2012.

[AYSS09]   Tuncer Can Aysal, Mehmet Ercan Yildiz, Anand D. Sarwate, and Anna Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal Processing*, 57(7):2748–2761, 2009.

[BBC+09]   Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 1061–1062, New York, NY, USA, 2009. ACM.

[BBS04]   Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load management and high availability in the medusa distributed stream processing engine. In *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 929–930, 2004.

[BE13]   Yahya Benkaouz and Mohammed Erradi. A distributed protocol for privacy preserving aggregation. In *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, pages 221–232, 2013.

[BFG+13]   Antoine Boutet, Davide Frey, Rachid Guerraoui, Arnaud Jégou, and Anne-Marie Kermarrec. WhatsUp Decentralized Instant News Recommender. In *IPDPS 2013*, Boston, United States, May 2013.

[BGM03]   M Bawa and H Garcia-Molina. Estimating aggregates on a peer-to-peer network. *submitted for . . .*, 2003.

[BGPS05]   S. Boyd, a. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: design, analysis and applications. *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, 3(C):1653–1664, 2005.

[BGPS06]   Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[BHOT05]   Vincent D. Blondel, Julien M. Hendrickx, Alex Olshevsky, and John N. Tsitsiklis. Convergence in multiagent coordination, consensus, and flocking. *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference, CDC-ECC '05*, 2005:2996–3000, 2005.

[BMT+05]   Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Minerva: Collaborative p2p search. In *Proceedings of the 31st international conference on Very large data bases*, pages 1263–1266. VLDB Endowment, 2005.

[BPFC12]   Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, and Ugur Cetintemel. C-mr: Continuously executing mapreduce workflows on multi-core processors. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 1–8, New York, NY, USA, 2012. ACM.

[BW01]   Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[CATV11]   Jordi Creus, Bernd Amann, Nicolas Travers, and Dan Vodislav. Roses: a continuous query processor for large-scale rss filtering and aggregation. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2549–2552. ACM, 2011.

[CBB⁺03]   Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[CCD⁺03]   Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[CDGL00]   Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Answering queries using views over description logics knowledge bases. *AAAI/IAAI*, 2000:386–391, 2000.

[CDK⁺03]   Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Request Permissions, December 2003.

[CDTW00]   Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM.

[CF05]   Raphaël Chand and Pascal Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In JoséC. Cunha and PedroD. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 1194–1204. Springer Berlin Heidelberg, 2005.

[CGM02]   Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 23–32. IEEE, 2002.

[CGM05]   Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for p2p systems. In *Agents and Peer-to-Peer Computing*, pages 1–13. Springer, 2005.

[Cod70]   E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[CPS]   Haowen Chan, Adrian Perrig, and Dawn Song. Secure Hierarchical In-Network Aggregation in Sensor Networks Categories and Subject Descriptors.

[DG04]   Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DGH+87]   Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC*, 1987.

[EFGK03]   Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[EGH+03]   Patrick T. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight Probabilistic Broadcast. *TOCS*, 21(4):341–374, 2003.

[GS04]   Jun Gao and Peter Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *In ICNP 04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP 04*, pages 239–250. IEEE Computer Society, 2004.

[GVB01]   Indranil Gupta, Robbert Van Renesse, and Kenneth P. Birman. Scalable fault-tolerant aggregation in large process groups. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 433–442, 2001.

[Hal01]   Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, December 2001.

[HR08]   Maya Haridasan and Robbert Van Renesse. Gossip-based distribution estimation in peer-to-peer networks. . . . *Workshop on Peer-to-Peer Systems (. . .* , 2008.

[IKT04]   Stratos Idreos, Manolis Koubarakis, and Christos Tryfonopoulos. P2p-diet: An extensible p2p service that unifies ad-hoc and continuous querying in super-peer networks. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 933–934. ACM, 2004.

[JK82]   D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, pages 164–169, New York, NY, USA, 1982. ACM.

[JM04]   M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. pages 102–109, 2004.

[JVG+07]   Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based Peer Sampling. *TOCS*, 25(3):1–36, 2007.

[KB+05]   Yoram Kulbak, Danny Bickson, et al. The emule protocol specification. *eMule project, http://sourceforge. net*, 2005.

[KBC+00]   John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[KKM13]   Konstantinos Karanasos, Asterios Katsifodimos, and Ioana Manolescu. Delta: Scalable data dissemination under capacity constraints. *PVLDB*, 7(4):217–228, 2013.

[KT13]   Anne-Marie Kermarrec and Peter Triantafillou. XL peer-to-peer pub/sub systems. 46(2):16:1–16:??, November 2013.

[KV98]   Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 205–213, New York, NY, USA, 1998. ACM.

[KvS07]   Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, October 2007.

[KYS+06]   Jayanthkumar Kannan, Beverly Yang, Scott Shenker, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung ju Lee. Smartseer: Using a dht to process continuous queries over peer-to-peer networks. In *In Proceedings of the 2006 IEEE INFOCOM*, 2006.

[LCP+05]   Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.

[LMD+11]   Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.

[LZRE14]   Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh. Redoop infrastructure for recurring big data queries. *Proc. VLDB Endow.*, 7(13):1589–1592, August 2014.

[LZRE15]   Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh. Shared execution of recurring workloads in mapreduce. *PVLDB*, 8(7):714–725, 2015.

[MJB04]   a. Montresor, M. Jelasity, and O. Babaoglu. Robust aggregation protocols for large-scale overlay networks. *International Conference on Dependable Systems and Networks, 2004*, 2004.

[MMGC02]   Athicha Muthitacharoen, Robert Morris, Thomer M Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[MPV06]   Vidal Martins, Esther Pacitti, and Patrick Valduriez. Survey of data replication in P2P systems. Research Report RR-6083, 2006.

[MS04]   Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, January 2004.

[MV06]   Ciamac C. Moallemi and Benjamin Van Roy. Consensus propagation. *IEEE Transactions on Information Theory*, 52(11):4753–4766, 2006.

[NOTZ03]   Wee Siong Ng, Beng Chin Ooi, K-L Tan, and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 633–644. IEEE, 2003.

[NRNK10]   L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, Dec 2010.

[Ora01]   Andrew Oram. *Peer-to-peer: harnessing the benefits of a disruptive technology*. " O'Reilly Media, Inc.", 2001.

[PC05]   O. Papaemmanouil and U. Cetintemel. Semcast: Semantic multicast for content-based data dissemination. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 242–253, 2005.

[PGES05]   Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[PPHH01]   Rachel Pottinger, Rachel Pottinger, Alon Halevy, and Alon Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2):182 – 198, 2001.

[RFH⁺01]   Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[SCSH12]   Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[SHLB08]   Hailong Sun, Jinpeng Huai, Yunhao Liu, and Rajkumar Buyya. Rct: A distributed tree for supporting efficient range and multi-attribute queries in grid computing. *Future Generation Computer Systems*, 24(7):631–643, 2008.

[SHLD11]   B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 348–355, July 2011.

[SMK⁺01]   Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[TATV11]   Jordi Creus Tomàs, Bernd Amann, Nicolas Travers, and Dan Vodislav. Roses: A continuous content-based query engine for rss feeds. In *Database and Expert Systems Applications*, pages 203–218. Springer, 2011.

[TB05]   Iradj Ouveysib Tolga Bektasa, Osman Oguza. Designing cost-effective content distribution networks. In *Computers and Operations Research*, October 2005.

[TIKR14]   Christos Tryfonopoulos, Stratos Idreos, Manolis Koubarakis, and Paraskevi Raftopoulou. Distributed large-scale information filtering. In Abdelkader Hameurlain, Josef Küng, and Roland Wagner, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems XIII*, volume 8420 of *Lecture Notes in Computer Science*, pages 91–122. Springer Berlin Heidelberg, 2014.

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

[VBV03]   Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21:164–206, 2003.

[VvS05]   Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 1143–1152, Berlin, Heidelberg, 2005. Springer-Verlag.

[XCTS14]   Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544, June 2014.

[YC84]   C. T. Yu and C. C. Chang. Distributed query processing. *ACM Comput. Surv.*, 16(4):399–433, December 1984.

[YP04]   Cong Yu and Lucian Popa. Constraint-based xml query rewriting for data integration. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 371–382, New York, NY, USA, 2004. ACM.

[ZDL⁺12]   Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: A fault-tolerant model for scalable stream processing. Technical Report UCB/EECS-2012-259, EECS Department, University of California, Berkeley, Dec 2012.

7   Webographie.

[w1w] *Google Alerts.*

....https://www.google.fr/alerts

[w2w] *Nest.*

....https://nest.com/

[w4w] *XQuery.*

....http://www.w3.org/TR/xquery/

[w5w] *XPath.*

....http://www.w3.org/TR/xpath20/

[w6w] *SparkQL.*

.....................................http://www.w3.org/TR/rdf-sparql-query/

[w7w] *Storm.*

....https://storm.apache.org/

[w8w] *Hadoop.*

. . . . .http://hadoop.apache.org/

[9] *MongoDB.*

. . . . .http://www.mongodb.org/

[10] *Oracle.*

. . . . .https://www.oracle.com/database/

[11] *Database Administrator's Guide : Scheduler Concepts.*

http://docs.oracle.com/cd/B19306_01/server.102/b14231/schedover.htm

Contents