Managing Views Over Streams
PhD First Year Report

Alasdair J G Gray
Supervisor: Dr. Werner Nutt

13 October 2003

Abstract

The aim of this PhD work is to create and maintain a hierarchy of views over a set of distributed data streams. This can be broken down into two objectives:

1. Query Planning and Replanning: creating and updating query plans whenever a new primary, or derived source of data is created.

2. Query Execution: define appropriate protocols for moving from an existing query plan to an updated one.

This work will feed into the query planning and execution performed within the R-GMA system.

Contents

Chapter 1

Introduction

The goal of this PhD work is first to assess current theoretical results on view based query processing and second, to refine and extend them where necessary to apply to distributed data streams.

We will develop this work as part of the Relational Grid Monitoring Architecture (R-GMA) system that is being developed within the European DataGrid project [Dat03]. DataGrid aims to develop a computational Grid for processing and analysing data. There are currently three groups for whom DataGrid is being developed: the partical physicists using the Large Hadron Collider at CERN, biological and medical image processing as part of exploration of genomes, and processing images captured in space of the Earth's atmospheric ozone.

R-GMA has been developed over the last two years by Heriot Watt University, IBM-UK, PPARC, Queen Mary University of London, Rutherford Appleton Laboratory, SZTAKI, and Trinity College Dublin. Recent papers about the R-GMA system include [BCC+02, BCC+03]. R-GMA has an open-source license, and can be downloaded from [RGM03].

In chapter 2 we will discuss related work and theory covering computational Grids, information integration, and data streams. Chapter 3 describes the R-GMA system. In chapter 4 we detail the problem of managing views over streams and discuss what a solution must achieve. Finally, chapter 5 describes how we intend to solve the problem of managing views over streams.

Chapter 2

Background

The proposed work draws upon various fields of research: computational Grids, information integration, and data streams. We will outline the results from these fields which will directly impact on managing views over data streams.

2.1 Grid Computing

We introduce the idea of computational Grids and discuss the problem of monitoring the status of a computational Grid. This will provide us with the environment for our data streams.

2.1.1 Computational Grids

A *computational Grid* is a collection of connected, geographically distributed computing resources belonging to several different organisations. Typically the resources are a mix of computers, storage devices, network bandwidth and specialised equipment, e.g. supercomputers or databases. Foster and Kesselman, who are leading the Globus Grid project [Glo03], gave the following definition:

> "A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [FK99]

A computational Grid provides instantaneous access to files, remote computers, software and specialist equipment. To a user, a Grid behaves like a single virtual supercomputer.

The idea of a computational Grid is based on a metaphor with the power Grid, which is used to supply electricity. In the same way as one plugs an electrical device into the power Grid and gets instant electrical power, a user should be able to "plug" into a computational Grid and gain instant computational power. The metaphor can be carried along further. Just as electricity in the power Grid can be provided by several companies, the computational resources in a computational Grid may be provided by several different organisations. Moreover, it must be possible to "charge" the user of the Grid for use of these resources. The payment may be in the form of money based on the amount that the resource was utilised, or by making their own resources available to other Grid users.

The concept of a computational Grid has existed since the mid 1990s and has grown out of the distributed and high performance computing communities. There have now been several projects to construct computational Grids to perform different tasks. These include:

- AstroGrid: aims to allow astronomers from around the UK to access astronomical databases through a standard interface. It will provide computational resources for processing data to enable a Virtual Observatory [Man01].
- Information Power Grid: under development at NASA to allow all their computational resources to be shared [JGN99]. These computational resources include equipment for experiments, e.g. mission critical software and supercomputers for running simulations.
- TeraGrid: aiming to build a Grid for scientific research capable of 20 teraflops of computation, distributed across 5 sites in the USA, with 1 petabyte of storage, and a network bandwidth of 40 Gigabits per second [Ber01].
- Globus Toolkit: aims to provide a generic infrastructure and tools for the creation of computational Grids [Glo03].

To make a computational Grid behave as a virtual computer requires various components that mimic the behaviour of a computer's operating system. The components of DataGrid, and their interactions, can be seen in figure 2.1 and are similar to those presented in [FKT01].

User Interface: allows a human user to submit jobs, e.g. "analyse the data from a physics experiment, and store the result".

Resource Broker: controls the submission of jobs, finds suitable available resources and allocates them to the job.

Logging and Bookkeeping: tracks the progress of jobs, informs users when jobs are completed, which resources were used, and how much they will be charged for the job.

Storage Element (SE): provides physical storage for data files.

Replica Catalogue: tracks where data is stored and replicates data files as required.

Computing Element (CE): performs the processing of jobs, taking data from storage elements.

Monitoring System: monitors the state of the components of the Grid and makes this data available to other components.

2.1.2 Grid Monitoring

The purpose of a Grid monitoring system is to make information about the status of a Grid available to users and to other components of the Grid. The following are typical use cases:

1. A resource broker needs to locate a computing element (CE), that has 5 CPUs available each with at least 200 MB of memory. The CE should have the right software installed, and the user must be authorised to use it. The throughput to a SE needs to be greater than 500 Mbps.

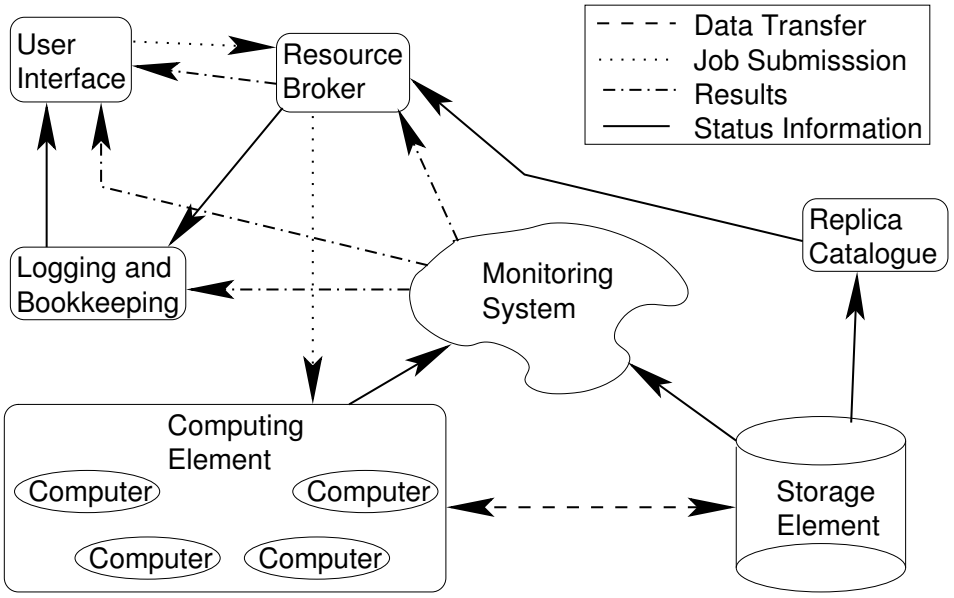

Figure 2.1: The major components of DataGrid.

2. A visualisation tool that is used by users to monitor the progress of their jobs needs to be updated whenever the status of a job changes.

3. Network administrators need to interrogate the past state of the network so that typical behaviour can be ascertained and anomalies identified.

Use case 1 shows that a Grid monitoring system must be able to cope with both static and streaming data. The number of CPUs that a computing element has will not change, so this is static data. The throughput of the network is changeable, depending on how heavily loaded the network is at any moment, so this is streaming data.

Use case 2 shows that the freshness of data is important. A user will want to know what the current state of their job is, not the state of it yesterday.

Use case 3 shows that some users will be interested in the past status of the Grid. The monitoring system must ensure that this information is available for data retrieval.

The Grid monitoring system will also have the usual design constraints of a distributed system, namely; scalability, robustness, performance, and security.

Grid Monitoring Architecture

The Grid Monitoring Architecture (GMA), proposed by Tierney *et al.* [TAG+00], is a general framework for monitoring Grids, and has been accepted as a standard by the Global Grid Forum [GGF03]. The framework consists of three main types of components:

Producers: A source of monitoring information, e.g. a sensor monitoring the amount of space left on a disk.

Consumers: A user of monitoring information, e.g. a resource broker trying to find a lightly loaded computing element.

Directory Service: Allows consumers to locate relevant producers of data.

The interaction of these components can be seen in figure 2.2. A producer informs the directory service of the kind of data it has to offer. A consumer contacts the directory service to discover which producers have data relevant to its needs. Communication links are then set up directly between the consumer and the relevant producers to acquire the data. Consumers also register with the directory service. This allows new producers to notify any consumers who will be interested in their data.

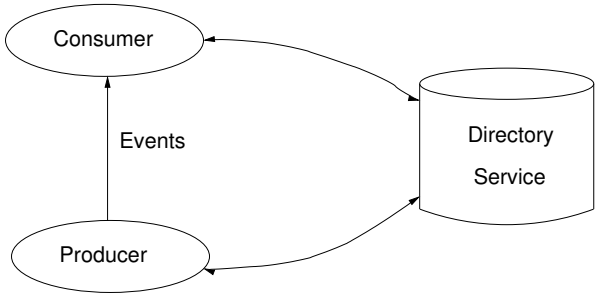

Figure 2.2: The Grid monitoring architecture

Intermediary components may be set up that consist of both a consumer and a producer. Intermediaries may be used to forward, broadcast, filter, aggregate, or archive data from other producers. The intermediary then makes this data available for other consumers from a single point in the Grid.

The GMA does not define a data model, query language, or a protocol for data transmission. Nor does it say what information should be stored in the directory service. There are no details of how the directory service should perform the task of matching producers with consumers.

Existing Grid Monitoring Systems

There are currently several Grid monitoring systems based on the GMA. These include AutoPilot [RVSR98], CODE [Smi02], and the Monitoring and Discovery Service (MDS) [CFFK01]. The most widely used at present is the MDS system which is part of the Globus Toolkit [Glo03].

The MDS[1] system uses the Lightweight Directory Access Protocol (LDAP) as a means of providing a name space, query language, and wire protocol for the GMA. This does mean that MDS suffers from the drawbacks of the LDAP query language,

- the name space must be designed carefully with common queries in mind;
- it is not possible to connect data from different parts of the name space, *cf. relational join operator.*

The MDS system uses a polling protocol and caches to maintain current values about the status of devices on the Grid. This means that there is some latency in the values as the devices are not continually pushing their values into the system.

[1]We discuss here MDS version 2. This is broadly similar to version 1, but has a decentralised LDAP hierarchy. Version 3 of MDS is currently being developed which will move away from using LDAP.

With further research, the drawbacks of MDS could be overcome. Systems based on the relational model using a suitable query language do not need to worry about the design of the name space and allow for the arbitrary joining of data. It would be interesting to link this with a push model, making use of data streams (section 2.3).

2.2 Information Integration

Information integration aims to allow a collection of distributed sources, e.g. a collection of e-commerce websites or a collection of relational databases, to be queried as if they were a single, *virtual database*. To make the collection of sources appear as a virtual database requires three areas to be addressed: source content description through a common schema, generating a query plan for the user posed query, and execution of the query plans over a set of distributed data sources.

A general information integration system architecture is presented in figure 2.3 [Kos00]. The Mediator component is responsible for parsing the user query posed against the virtual database. It will then generate suitable queries in terms of the data sources available. An execution plan is then generated managing the retrieval of data from the sources and combining the answer sets. The schema of the virtual database is stored in the Catalogue along with a mapping relating each source with the schema of the virtual database. Each source must be encapsulated in a Wrapper which gives the impression that the source is a relational database.

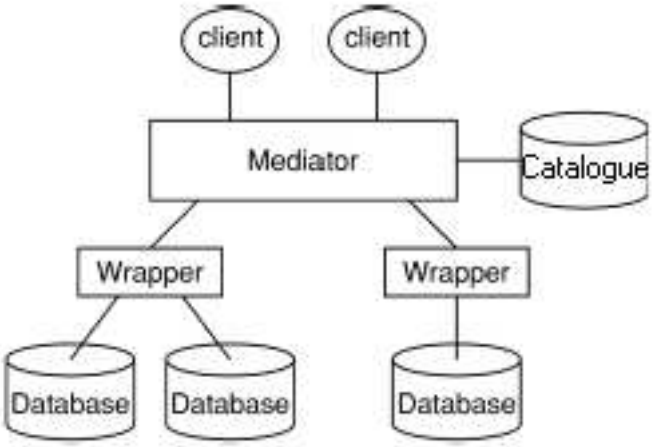

Figure 2.3: Architecture of an information integration system.

2.2.1 Schema Level Integration

As shown in the general architecture of an information integration system (figure 2.3), the virtual database is made up of a collection of sources each appearing as a separate relational database. Each source is managed independently, and thus has its own *local schema* to which the data conforms. The sources are integrated into the virtual database by means of a *global schema*. Ullman, in his paper [Ull97], identified two types of schema level integration which were named by Levy [Lev00] as "global as view" and "local as view".

First we shall consider the "global as view" approach. Consider the global schema

$$\mathsf{Person}(\mathsf{name}, \mathsf{address}, \mathsf{age}, \mathsf{job}),$$

to represent information about a person, storing their name, address, age and job title. Say there are two data sources in the system,

1. A company database storing information about employees. This contains the relations

$$\mathsf{Employee}(\mathsf{NI\#}, \mathsf{name}, \mathsf{age}, \mathsf{job}) \tag{2.1}$$
$$\mathsf{Address}(\mathsf{NI\#}, \mathsf{address}), \tag{2.2}$$

   where relation (2.1) stores the name, age, and job title of an employee using their national insurance number as a key. Relation (2.2) stores addresses, relating each address to an employee based on the national insurance number.

2. A university's database storing information about the students, containing the relation

$$\mathsf{Student}(\mathsf{matric\#}, \mathsf{name}, \mathsf{address}, \mathsf{age}),$$

   which stores the name, address, and age of the students based on their matriculation number.

The information integration system needs to store mappings in the Catalogue relating the local relations to the global schema. In the global as view approach this is achieved by storing view definitions that have the global relations as view predicates. We shall use Datalog notation to express the views. Datalog notation has predicate symbols, variable symbols, and constants. For our example the view definitions would be

$$\begin{aligned} \mathsf{Person}(name, address, age, job) \;\leftarrow\; & \mathsf{Employee}(NI\#, name, age, job)\ \&\ \\ & \mathsf{Address}(NI\#, address) \qquad (2.3)\\ \mathsf{Person}(name, address, age, \mathsf{'student'}) \;\leftarrow\; & \mathsf{Student}(\_, name, address, age). \qquad (2.4) \end{aligned}$$

View (2.3) provides a tuple to the global relation Person when there is a tuple in Employee and a tuple in Address which contain the same value for the attribute $NI\#$. View (2.4) provides a tuple to the global relation Person when there is a tuple in Student. The "_" is interpreted as a variable that appears only once and whose value is not output.

Query planning in the global as view approach is reasonably straightforward. The query posed over the global schema is transformed into queries over the data sources by substituting the view definitions for each occurrence of a predicate. For example, consider the query

$$\mathsf{q}(address) \;\leftarrow\; \mathsf{Person}(\mathsf{'john\ smith'}, address, age, job),$$

which is asking for the address of all the people called 'john smith'. By replacing the global relation occurring in the query with the view definitions (2.3) and (2.4) we generate the following queries which the Mediator must pose against the sources

$$\begin{aligned} \mathsf{q}_1(address) \;&\leftarrow\; \mathsf{Employee}(NI\#, \mathsf{'john\ smith'}, age, job)\ \&\ \mathsf{Address}(NI\#, address)\\ \mathsf{q}_2(address) \;&\leftarrow\; \mathsf{Student}(matric\#, \mathsf{'john\ smith'}, address, age). \end{aligned}$$

The execution of these queries against the data sources must be co-ordinated by the Mediator. Techniques for co-ordinating the execution of these queries against distributed sources are discussed in section 2.2.6.

We now turn our attention to the local as view approach. Consider the global schema

$$\mathsf{Address}(\mathsf{name}, \mathsf{street}, \mathsf{city}) \tag{2.5}$$

$$\mathsf{Phone}(\mathsf{name}, \mathsf{number}). \tag{2.6}$$

Relation (2.5) gives the name and address of someone, and relation (2.6) gives the name and phone number. Say there are two data sources in the system

1. A phone book with the local schema

   $$\mathsf{PhoneBook}(\mathsf{name}, \mathsf{city}, \mathsf{number}),$$

   which provides the name, city, and telephone number of someone.

2. An address book with the local schema

   $$\mathsf{AddressBook}(\mathsf{name}, \mathsf{street}, \mathsf{city}),$$

   which stores the name, the street that they live on, and the city of someone.

Each local schema is related to the global schema by a view definition, stored in the Catalogue. The queries of the view definition will be posed over the global schema. For our example, this would give us the following view definitions.

$$\begin{aligned} \mathsf{PhoneBook}(name, city, number) \leftarrow\ & \mathsf{Address}(name, street, city)\ \& \\ & \mathsf{Phone}(name, number) \end{aligned} \tag{2.7}$$

$$\mathsf{AddressBook}(name, street, city) \leftarrow \mathsf{Address}(name, street, city) \tag{2.8}$$

The view definitions give the properties that tuples must have, but they do not guarantee to provide all tuples that conform to the definition. There is also no guarantee of consistency between the sources.

A query that may be posed against this system could be

$$\mathsf{q}(city) \leftarrow \mathsf{Address}('\mathsf{john\ smith}', street, city), \tag{2.9}$$

which would be locating the city of anyone called 'john smith'. It is not immediately clear how to generate an answer to this query, as the tuples are stored in the data sources which are related to the global schema by the view definitions (2.7) and (2.8). In the next four subsections we shall discuss the theory needed to generate answers to query (2.9).

2.2.2 Answering Queries Using Views

In the local as view approach, query planning is not as straightforward as in the global as view approach. This is due to the fact that the predicates in the query appear in the body of the view definitions and therefore the unfolding approach that we used earlier is no longer applicable.

We first need to understand what an answer to the query is. Formally, we have a globally defined schema $\mathcal{G}$ and a set of source relations S_i which provide the data for the system. There is a set of *declarations* Δ containing one declaration for each source. Each declaration is of one of the following two forms:

$$S_i \subseteq v_i \qquad \text{(Inclusion declaration)} \tag{2.10}$$

$$S_i = v_i \qquad \text{(Equality declaration)}, \tag{2.11}$$

where v_i is a view predicate that is defined by a query on the global schema $\mathcal{G}$. Intuitively, the inclusion declaration (2.10) says that the source S_i provides some of the answers meeting the view definition v_i. The equality declaration (2.11) says that the source S_i provides all of the answers meeting the view definition v_i.

A user poses a query q over the global schema $\mathcal{G}$. However, there is no data for $\mathcal{G}$, the data is contained in the data sources S_i which are related to the global schema through the declarations Δ. To return answers to the query q we must use the views v_i which relate the set of sources S_i to the global schema $\mathcal{G}$.

Databases Consistent with the Declarations

Since the user query is posed over the global schema, but there is only data in the sources, we must consider whether it is possible to "construct" the part of the database state from the view definitions that is needed to answer the query. The content of the sources, i.e. the S_i, do not uniquely determine an instance of $\mathcal{G}$. There can be infinitely many databases that conform to the set of declarations.

We denote an instance of the relational sources with $\mathcal{I}$. The set of tuples present in the source relation S_i at the instance $\mathcal{I}$ is given by $S_i^{\mathcal{I}}$. An instance of the global schema is represented by $\mathcal{D}$ which we require to be consistent with the declarations and the instance $\mathcal{I}$. For $\mathcal{D}$ to be consistent with the declarations means that the tuples in the sources meet the view definitions.

Definition 1 (Consistent Database Instance) *Let $\mathcal{I}$ be an instance of the sources and $\mathcal{D}$ be an instance of $\mathcal{G}$. We say that $\mathcal{D}$ is* consistent *with*

- *an equality declaration $S_i = v_i$ if $S_i^{\mathcal{I}} = v_i^{\mathcal{D}}$;*
- *an inclusion declaration $S_i \subseteq v_i$ if $S_i^{\mathcal{I}} \subseteq v_i^{\mathcal{D}}$;*
- *a set of declarations Δ if $\mathcal{D}$ is consistent with each of the declarations in the set.*

Answers to a Query

The question then arises, when a user poses a query q against the global schema, which tuples should be returned as answers, and therefore which sources should be contacted? The tuples which should be returned are those that are *certain answers* [AD98], i.e. those tuples which every database instance, consistent with an instance of the set of source declarations, would return as an answer.

Definition 2 (Certain Answers) *A tuple $\bar{t}$ is a* certain answer *for a query q with respect to an instance $\mathcal{I}$ if for all consistent databases $\mathcal{D}$*

$$\bar{t} \in q^{\mathcal{D}}.$$

We denote the set containing all certain answers for query q with respect to the instance $\mathcal{I}$ as $\mathsf{cert}(q, \mathcal{I})$.

The problem of answering queries using views is then a case of computing the set of all certain answers from the source relations. The set of certain answers is

$$\mathsf{cert}(q, \mathcal{I}) = \bigcap_{\substack{\mathcal{D} \\ \text{consistent} \\ \text{with } \mathcal{I}}} q^{\mathcal{D}},$$

but it is not clear how to compute this.

We will now go on to discuss how the set of certain answers can be generated. We will begin by looking at the properties of the query posed against the global schema (section 2.2.3). We will then present techniques for transforming the query posed over the global schema into queries over the data sources in the system (section 2.2.4). To conclude our discussion of the local as view approach, we will briefly mention some of algorithms that perform this task and generate the "best" local queries for a given global query (section 2.2.5).

2.2.3 Properties of Queries

First we shall formalise the concept of a database $\mathcal{D}$. A database $\mathcal{D}$ is a collection of ground atoms. A ground atom is a predicate symbol, i.e. a relation's name, and a tuple of constants. For example, the data source PhoneBook(name, city, number) might contain the ground atom,

$$\mathsf{PhoneBook}(\text{'john smith'}, \text{' Edinburgh'}, 240\ 8473).$$

A query q is posed against a database $\mathcal{D}$ and the set of tuples generated as the answer is denoted as $q^{\mathcal{D}}$.

Below, we will give formal definitions for query equivalence and query containment. Intuitively, two queries are said to be *equivalent* if they return the same set of tuples as answers over all databases. Alternatively, the set of tuples generated by one of the queries may be a subset of the tuples generated by the other. This is called *containment*.

Definition 3 (Query Equivalence) *A query q_1 is* equivalent *to query q_2, written $q_1 \equiv q_2$, iff for all databases $\mathcal{D}$ we have*

$$q_1^{\mathcal{D}} = q_2^{\mathcal{D}}.$$

Definition 4 (Query Containment) *A query q_1 is* contained *in query q_2, written $q_1 \subseteq q_2$, iff for all databases $\mathcal{D}$ we have*

$$q_1^{\mathcal{D}} \subseteq q_2^{\mathcal{D}}.$$

Conjunctive Queries

Conjunctive queries are an interesting class of queries. They can be used to represent a select-project-join query and equivalence and containment of queries can be decided.

A relational conjunctive query is defined by a rule of the form

$$q(\bar{X}) \leftarrow p_1(\bar{X}_1) \,\&\, \ldots \,\&\, p_n(\bar{X}_n).$$

The head of the conjunctive query is the atom $q(\bar{X})$ where $\bar{X}$ is a tuple of terms made up of variables and constants. The body of the conjunctive query is made up of the atoms $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ where each $\bar{X}_i$ is a tuple of terms. A conjunctive query is *safe* if and only if the variables that appear in the head also appear in at least one of the atoms in the body. For example

$$\begin{aligned} \mathsf{q}(\text{'john smith'}, city, number) \leftarrow\ & \mathsf{Address}(\text{'john smith'}, street, city) \,\& \\ & \mathsf{Phone}(\text{'john smith'}, number), \end{aligned} \tag{2.12}$$

is a safe conjunctive query returning the atom $\mathsf{q}(\text{'john smith'}, city, number)$ from the ground atoms $\mathsf{Address}(\text{'john smith'}, street, city)$ and $\mathsf{Phone}(\text{'john smith'}, number)$.

The tuples returned as the head of a conjunctive query, when posed over a database $\mathcal{D}$, are the tuples $\delta(\bar{X})$, where δ is an assignment of the variables in the conjunctive query that satisfy each atom in the body of the query.

Definition 5 *Let* $\mathsf{q}(\bar{X}) \leftarrow \mathsf{p}_1(\bar{X}_1) \,\&\, \ldots \,\&\, \mathsf{p}_n(\bar{X}_n)$ *be a conjunctive query. An assignment of variables* δ satisfies *a conjunctive query w.r.t. a database* $\mathcal{D}$ *iff we have for each atom in the body,*

$$p_i(\delta\bar{X}_i) \in \mathcal{D}.$$

Whenever δ is a satisfying assignment of variables then $q(\delta\bar{X})$ is an answer to the conjunctive query. Assignments can be extended to constants in such a way that each constant is mapped to itself. For our example conjunctive query (2.12), if we posed the query over the database containing the atoms

$$\begin{aligned} &\mathsf{Address}(\text{'john smith'}, \text{' high street'}, \text{' Edinburgh'}) \\ &\mathsf{Phone}(\text{'john smith'}, 240\ 8473), \end{aligned}$$

then the assignment

$$\begin{aligned} \text{'john smith'} &\mapsto \text{'john smith'} \\ \mathsf{street} &\mapsto \text{'high street'} \\ \mathsf{city} &\mapsto \text{'Edinburgh'} \\ \mathsf{number} &\mapsto 240\ 8473 \end{aligned}$$

would result in the atom

$$\mathsf{q}(\text{'john smith'}, \text{' Edinburgh'}, 240\ 8473),$$

as an answer to the query with respect to the database.

Two conjunctive queries can have the properties of containment and equivalence. Whether a conjunctive query q_1 is contained by a conjunctive query q_2, can be decided by finding a homomorphism from q_2 to q_1. Deciding whether such a homomorphism exists is an NP-complete problem [CM77]. For equivalence, we must also find a homomorphism from the terms in q_1 to the terms in q_2.

When the conjunctive queries can contain comparison operations, e.g. $number \geq 240\ 0000$, then a case analysis based on the comparison operations must be performed. A homomorphism from q_2 to q_1 for each case must be found [Klu88]. An alternatively view of this is that q_1 can be equivalently rewritten as a disjunction of possibly exponentially many conjunctive queries, one disjunct for every case. Since there could be an exponential number, the complexity of deciding whether such homomorphisms exist is complete for Π_2^{P} [vdM92].

Research continues to be conducted into the properties of queries, for example more recently the ideas of containment and equivalence have been extended to aggregate queries [CNS01].

Monotonic Queries

An important property of a query is monotonicity. We give a formal definition below. Intuitively, a query is *monotonic* if the set of answers generated by the query remains the same or increases if the database has additional tuples. Examples of a monotonic query are select-project-join queries without negation.

Definition 6 (Monotonic Query) *A query q is* monotonic *if $q^{\mathcal{D}_1} \subseteq q^{\mathcal{D}_2}$ whenever $\mathcal{D}_1 \subseteq \mathcal{D}_2$.*

2.2.4 Rewriting Queries Using Views

Rewriting queries using views is a technique to compute certain answers for a query q posed over a global schema $\mathcal{G}$. We assume that S_i is a data source which is related to $\mathcal{G}$ by a view definition v_i. There is exactly one view definition for each source. The query q can then be rewritten as a set of queries posed over the data sources by using the view definitions. An overview of the techniques can be found in [Hal01].

To ensure that we only retrieve certain answers from the sources S_i we have to compare the queries that we pose over the sources with the original query posed over $\mathcal{G}$. Therefore, we extend our notion of equivalent and contained queries to allow our databases to be extended by a set of view definitions,

$$\mathcal{V} = \{v_1, \ldots, v_k\},$$

where each v_i is defined by a query over $\mathcal{G}$ of the form

$$v_i(\bar{X}_i) \ \leftarrow \ B_i,$$

where the only predicates found in B_i, the body of the query, are those from $\mathcal{G}$.

Properties of Queries With Respect to Views

We begin by extending our notion of a database instance $\mathcal{D}$ to include the set of views $\mathcal{V}$. For every instance $\mathcal{D}$ of $\mathcal{G}$ we construct an instance $\mathcal{D}^{\mathcal{V}}$ for the relations of $\mathcal{G}$ with the views in $\mathcal{V}$. Formally, this is the set of atoms,

$$\mathcal{D}^{\mathcal{V}} = \mathcal{D} \cup \{\, v_i(\bar{X}_i) \mid \bar{X}_i \in v_i^{\mathcal{D}}, \text{ and } 1 \leq i \leq k \,\}.$$

This implies that $\mathcal{D}^{\mathcal{V}}$ consists of the relation instances in $\mathcal{D}$ together with instances $v_i^{\mathcal{D}}$ for every view predicate v_i.

We now extend our earlier definitions of query equivalence and query containment to incorporate database instances $\mathcal{D}$ extended by a set of views $\mathcal{V}$.

Definition 7 (Equivalent Queries w.r.t. $\mathcal{V}$) *A query q_1 is* equivalent *w.r.t. a set of views $\mathcal{V}$ to a query q_2, written $q_1 \equiv_{\mathcal{V}} q_2$, iff for all databases $\mathcal{D}$ we have*

$$q_1^{\mathcal{D}^{\mathcal{V}}} = q_2^{\mathcal{D}^{\mathcal{V}}}.$$

Definition 8 (Query Containment w.r.t. $\mathcal{V}$) *A query q_1 is* contained *w.r.t. a set of views $\mathcal{V}$ in a query q_2, written $q_1 \subseteq_{\mathcal{V}} q_2$, iff for all databases $\mathcal{D}$ we have*

$$q_1^{\mathcal{D}^{\mathcal{V}}} \subseteq q_2^{\mathcal{D}^{\mathcal{V}}}.$$

Rewritings

Each data source S_i is identified by a view definition v_i. By evaluating the views over the data sources, we are able to answer queries that contain the predicates of the view. By applying the techniques of rewriting queries using views, we can transform the query q posed over $\mathcal{G}$ into a rewritten query r posed over the set of views $\mathcal{V}$. The rewriting r is said to be an equivalent rewriting of q if $r \equiv_{\mathcal{V}} q$ holds. Likewise, it is a contained rewriting of q if $r \subseteq_{\mathcal{V}} q$ holds.

We can check whether a query r that contains view predicates is a rewriting of the query q posed over $\mathcal{G}$ by a technique known as *unfolding*. To unfold r involves replacing all occurrences of view predicates in the body of r by their definition. Care must be taken with the existential variables in the body of the view definitions. We denote the unfolded query as r^u which will be a query over $\mathcal{G}$. We can then check whether r^u is contained in, or equivalent to, q by finding a homomorphism (section 2.2.3).

More precisely, suppose we have in the body of r an atom $v(\bar{t})$ and suppose

$$v(\bar{X}) \leftarrow B,$$

is the definition rule of $v(\bar{X})$. Then we unfold the atom $v(\bar{t})$ in r by replacing it with the instantiated body σB where σ is the substitution that replaces variable x_i in $\bar{X}$ with the term t_i in $\bar{t}$.

The following proposition shows that the process of unfolding does not affect the equivalence w.r.t. $\mathcal{V}$, or the containment w.r.t. $\mathcal{V}$ of the rewriting. Intuitively, this is due to the fact that the unfolded query is posed only over $\mathcal{G}$ so the containment can be over the database only. The unfolding itself does not affect the query as we are replacing a view predicate by its body which is used to generate the tuples in the view.

Proposition 1 *Let q be a query over $\mathcal{G}$, r be a query over $\mathcal{G}$ and $\mathcal{V}$, and r^u be the unfolding of r. Then*

- $r \equiv_{\mathcal{V}} q$ *iff* $r^u \equiv q$;
- $r \subseteq_{\mathcal{V}} q$ *iff* $r^u \subseteq q$.

We will now show that a contained rewriting r for the query q will generate a subset of the set of certain answers $\mathsf{cert}(q, \mathcal{I})$. The complete set of certain answers can be computed if we have an equivalent rewriting. We denote the set of tuples generated by posing the rewriting over an instance of the sources as $r^{\mathcal{I}}$. Theorem 1 states the case when the set of declarations are all equality declarations, while theorem 2 states the case in general.

Theorem 1 *Let q be a query posed over the global schema $\mathcal{G}$ and suppose that the set of declarations Δ consists only of equality declarations.*

- *If r is a* contained *rewriting of q then*

$$r^{\mathcal{I}} \subseteq \mathsf{cert}(q, \mathcal{I}).$$

- *If r is an* equivalent *rewriting of q then*

$$r^{\mathcal{I}} = \mathsf{cert}(q, \mathcal{I}).$$

Proof. By our assumption, declarations are of the form $S_i = v_i$. To prove our claim when r is a contained rewriting of q, we have to show that

$$r^{\mathcal{I}} \subseteq \bigcap_{\substack{\mathcal{D} \\ \text{consistent} \\ \text{with } \mathcal{I}}} q^{\mathcal{D}}.$$

To show this inclusion, it suffices to show that for all databases $\mathcal{D}$ that are consistent with instance $\mathcal{I}$, we have

$$r^{\mathcal{I}} \subseteq q^{\mathcal{D}}.$$

Now let $\mathcal{D}$ be such a database. Then we have $r^{\mathcal{I}} = r^{\mathcal{D}^{\mathcal{V}}}$ by the fact that we only have equality declarations. Since r is a contained rewriting of q it follows that $r^{\mathcal{D}^{\mathcal{V}}} \subseteq q^{\mathcal{D}^{\mathcal{V}}}$ and because the query q is posed over the global schema $\mathcal{G}$ we have $q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}$. Therefore, it follows

$$r^{\mathcal{I}} = r^{\mathcal{D}^{\mathcal{V}}} \subseteq q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}.$$

To prove our claim when r is an equivalent rewriting of q, we must show that

$$r^{\mathcal{I}} = \bigcap_{\substack{\mathcal{D} \\ \text{consistent} \\ \text{with } \mathcal{I}}} q^{\mathcal{D}}.$$

To show this equality, it suffices to show that for all databases $\mathcal{D}$ that are consistent with instance $\mathcal{I}$, we have

$$r^{\mathcal{I}} = q^{\mathcal{D}}.$$

Now let $\mathcal{D}$ be such a database. Then we have $r^{\mathcal{I}} = r^{\mathcal{D}^{\mathcal{V}}}$ by the fact that we only have equality declarations. Since r is an equivalent rewriting of q it follows that $r^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}^{\mathcal{V}}}$ and because the query q is posed over the global schema $\mathcal{G}$ we have $q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}$. Therefore, it follows

$$r^{\mathcal{I}} = r^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}.$$

□

Theorem 2 *Let q be a query over the global schema $\mathcal{G}$ and Δ is a set of declarations. If r is a rewriting of q, and r is a monotonic query, then*

$$r^{\mathcal{I}} \subseteq \mathsf{cert}(q, \mathcal{I}).$$

Proof. By our assumption that Δ may contain some inclusion declarations. To prove our claim when r is a rewriting of q, we have to show that

$$r^{\mathcal{I}} \subseteq \bigcap_{\substack{\mathcal{D} \\ \text{consistent} \\ \text{with } \mathcal{I}}} q^{\mathcal{D}}.$$

To show this inclusion, it suffices to show that for all databases $\mathcal{D}$ which are consistent with instance $\mathcal{I}$, we have

$$r^{\mathcal{I}} \subseteq q^{\mathcal{D}}.$$

Now let $\mathcal{D}$ be such a database. Then due to the fact $\mathcal{D}$ is consistent with $\mathcal{I}$, we have

$$S_i^{\mathcal{I}} \subseteq v_i^{\mathcal{D}_v}.$$

Since r is a monotonic query, we have $r^{\mathcal{I}} \subseteq r^{\mathcal{D}^{\mathcal{V}}}$. Since r is a rewriting of q it follows that $r^{\mathcal{D}^{\mathcal{V}}} \subseteq q^{\mathcal{D}^{\mathcal{V}}}$ and because the query q is posed over the global schema $\mathcal{G}$ we have $q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}$. Therefore, it follows

$$r^{\mathcal{I}} \subseteq r^{\mathcal{D}^{\mathcal{V}}} \subseteq q^{\mathcal{D}^{\mathcal{V}}} = q^{\mathcal{D}}.$$

□

2.2.5 Logical Query Planning

We return now to our local as view example where we had the global query presented as query (2.9) over the global relations of (2.5) and (2.6) with the set of view declarations comprising views (2.7) and (2.8). Using the techniques of rewriting queries using views we can generate two possible rewritings of query (2.9)

$$\mathsf{r}_1(city) \leftarrow \mathsf{PhoneBook}(\text{'john smith'}, city, \_) \tag{2.13}$$

$$\mathsf{r}_2(city) \leftarrow \mathsf{AddressBook}(\text{'john smith'}, \_, city). \tag{2.14}$$

If the declarations relating the sources to the global schema are all inclusion declarations then to generate the fullest subset of $\mathsf{cert}(q, \mathcal{I})$ we must take the union of all the answer sets from each data source. For our example we would execute queries (2.13) and (2.14) and combine their answer sets. Due to the results of Grahne and Mendelzon [GM99], we will generate all certain answers.

In general, there will be several rewritings for a given query. For an information integration system, we are interested in contacting the minimum number of sources in order to generate the fullest possible answer. This is achieved by first discarding any rewritings that are contained in other rewritings. For example, if we had the rewriting

$$\mathsf{r}_3(city) \leftarrow \mathsf{AddressBook}(\text{'john smith'}, \_, \text{'Edinburgh'})$$

then it could be discarded as it is contained in r_2. Any tuples returned as certain answers by r_3 would returned as certain answers by r_2.

The rewritings can then be checked for containment w.r.t. $\mathcal{V}$. Again, we would only keep the most general rewritings. Consider adding the source

$$\mathsf{CityBook}(\mathsf{name}, \mathsf{city}) \leftarrow \mathsf{Address}(\mathsf{name}, \mathsf{street}, \mathsf{city}),$$

to our example. We would then get the additional rewriting

$$\mathsf{r}_4(\mathit{city}) \leftarrow \mathsf{CityBook}('\mathsf{john\ smith}', \mathit{city}).$$

Both r_1 and r_2 are contained in this rewriting w.r.t. $\mathcal{V}$. We can therefore generate the fullest possible certain answer set by only using r_4.

There are several algorithms for finding which rewritings to use in order to generate the largest possible certain answer set from the minimum number of sources. The two most notable ones are the *Bucket* algorithm [LRO96] and the *MiniCon* algorithm [PH01] which is an enhancement on the Bucket algorithm. The output of both of these algorithms is a query plan relating to the data sources.

2.2.6 Query Execution

The techniques for the organisation of the execution of a query plan over a set of distributed data sources is the same for both the global as view and the local as view approaches to schema level integration. An overview of techniques for distributed query execution are presented in [Kos00].

The techniques are based on the standard architecture for an information integration system (figure 2.3). The Wrappers inform the Mediator of the querying capabilities of their source, and possibly cost estimates for executing queries. The Mediator, based on this information, can then pass the Wrappers suitable subqueries in order to execute the query plan as efficiently as possible. The Mediator may have additional cost models, along with the capability to perform certain operations such as joins across sources, to aide the efficient execution of queries.

2.3 Data Streams

A data stream is an append only, time varying, unbounded, sequence of data. This data can be thought of as tuples complying to some relational schema. Streams of data appear in many different situations, e.g. financial applications, sensor networks, monitoring information. The properties of a data stream are

- it is continually changing;
- it is an append only sequence of values;
- the data can potentially be unordered;
- there is no upper bound to the amount of data;
- data must be processed immediately, then archived or discarded as the next value will be following;

- users are generally interested in the most up to date value possible, e.g. generally interested in the current temperature, not the temperature five minutes ago when enquiring about the current temperature.

A general overview of data streams and the issues involved can be found in [BBD+02].

2.3.1 Data Stream Management Systems.

A traditional database management system (DBMS) is not suitable for the management of data streams due to their nature. A DBMS allocates storage by knowing the size of a given relation but a data stream is unbounded. Through transaction processing protocols, a DBMS provides a consistent database to the user through a series of operations, making up the transaction. However, when dealing with data streams the user is often interested in the very latest values, so the queries being posed and updates to the system should not hold up inserts to the streams. The data in the stream must be dealt with immediately when it arrives, as freshness of data is a key issue. Therefore, techniques for storing, managing, and querying data streams are being developed with the nature of streams in mind.

Some centralised data stream management systems (DSMS) have already been developed, and some preliminary systems have been implemented, such as STREAM [BBD+02], Aurora [CCC+02], Tribeca [Sul96] and AIMS [Sha03]. They support querying and management of relations, akin to a relational database management system, only these relations may be either streaming or static.

To store the entire history of a data stream would require infinite storage capacity. So, techniques for reducing the size of the history have been developed for data streams. These include,

- synopsises or sketches [AMS96, FM83]: these provide periodic summaries of the stream, e.g. store the stream for a day, then take suitable averages and discard the data.
- random samples [AGP00, AGPR99, CMN99]: a snapshot of the stream. Instead of storing the entire stream, a sample of the stream is stored.
- histograms [IP99, PG99]: a summary structure designed to capture the distribution of values in the stream.

All of these techniques lead to approximate answers to a query as we no longer have the actual data stream to query.

2.3.2 Querying Data Streams

The language most commonly used for querying a relational database is SQL. This language is designed to allow the user to describe the data they are interested in, based on the schema, as opposed to how to retrieve the data. However, SQL was designed for static relations. This is particularly apparent when you consider aggregate queries, e.g. the following SQL query returns the average salary for each type of job.

```
SELECT   AVG(salary)
FROM     EMPLOYEE
GROUP BY jobType
```

This would be difficult to process over a data stream as it requires

1. the data in the `EMPLOYEE` relation to be sorted into groups based on the employee's job,
2. for each group the mean of the salary is taken.

A data stream could be split into groups as it arrives, but since it is unbounded the mean could not be taken as you do not know when you have seen all the data for each group.

An abstract semantics for a query language over data streams has been proposed and implemented by Arasu *et al.* [ABW03]. The main idea of the work is to convert between streams and relations (figure 2.4), which are both defined formally. We can then perform any relational operation to the stream when it is a relation and then convert the derived relation into a data stream. Through these transformations between data streams and relations we can apply a standard relational query language, e.g. SQL, to data streams. We will now present Arasu *et al.*'s abstract semantics.

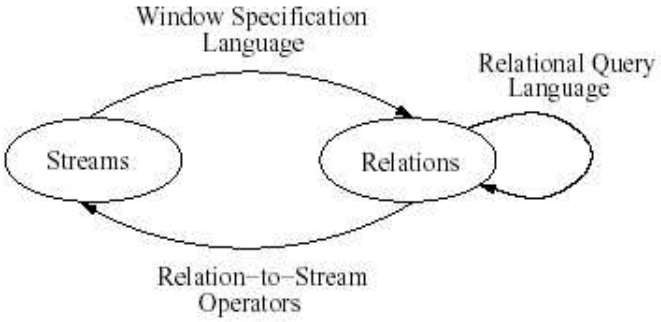

Figure 2.4: Converting between relations and data streams.

Streams and Relations

We begin by giving the definitions of a data stream and a relation from [ABW03]. These definitions assume that there is a global, discrete, ordered time domain. However, the timestamp from the time domain is not part of the schema. The timestamp is an additional attribute of a data stream ensuring that the values of the stream are linearly ordered.

Definition 9 (Data Stream) *A* data stream S *is a bag of elements* $\langle s, \tau \rangle$*, where* s *is a tuple belonging to the schema of the stream and* $\tau \in \mathcal{T}$ *is the timestamp of the element.*

Definition 10 (Relation) *A* relation R *is a mapping from* $\mathcal{T}$ *to a finite but unbounded bag of tuples, where each tuple belongs to the schema of the relation.*

From the definitions we can say element $\langle s, \tau \rangle \in S$ if tuple s arrives on stream S at time τ. Similarly, the elements of S up to time τ_0 are the elements of S with a timestamp τ such that $\tau \leq \tau_0$. A relation R defines an unordered bag of tuples at any time instant τ, denoted $R(\tau)$. The bag is unordered as a relation may be generated as the result of a relational algebra operation on another relation.

Mapping Operators

We need to be able to perform three types of mappings. This allows us to move between data streams and relations. The types of mapping are

Stream-to-Relation: Takes a data stream S as input and generates a relation R with the same schema as S.

Relation-to-Relation: Takes any number of relations $R_1, \ldots, R_n$ as input and generates a relation R, e.g. a relational join operation.

Relation-to-Stream: Takes a relation R as input and generates a data stream S with the same schema as R.

With these mappings it is possible to have a data stream generated by a sensor, perform a join with some static relation, and output the result of the join as a new data stream.

Abstract Semantics

The abstract semantics presented is based on three building blocks, corresponding to the three types of mapping operators.

1. Any relational query language. This provides the relation-to-relation mappings. In practise a fragment of SQL is commonly used.

2. A *window specification language*. Used to extract tuples from the data streams, i.e. provides the stream-to-relation mappings. In practise, there are three ways in which to specify the window,

   Time-Based Windows: Returns all the tuples that have a timestamp in some time-interval T. The output relation R generated by a time-interval of between now and T is defined as,

   $$R(\tau) = \{\, s \mid \langle s, \tau' \rangle \in S \wedge (\tau' \leq \tau) \wedge (\tau' \geq max\{\tau - T, 0\}) \,\}$$

   Tuple-Based Windows: Returns the last N tuples of the stream S. Due to the bag semantics, this window mechanism can be non-deterministic, as tuples get thrown away if they have the same, smallest timestamp.

   Partitioned Windows: This is similar to the SQL `GROUP BY` clause. The stream is split into groups based on some attributes, and the last N tuples in each group is returned.

3. Relation-to-stream operators. Used to convert derived relations into data streams. In the work at Stanford, they have only needed to define three such operators,

   Relation Stream: Returns a stream consisting of all the tuples present in the relation R at the time τ.

   Insert Stream: Returns a stream of tuples that have been added to the relation R, i.e. it will consist of the tuples which meet the condition $R(\tau) \setminus R(\tau - 1)$.

   Delete Stream: Returns a stream of tuples that used to be in the relation R, i.e. it will consist of the tuples which meet the condition $R(\tau - 1) \setminus R(\tau)$.

Remarks About Abstract Semantics

There is essentially no difference between the definitions given above for a data stream and a relation. For example, consider the data stream

$$\{\, \langle (a, b, c), 5 \rangle \; \langle (a, b, c), 5 \rangle \,\}.$$

When presented as a relation, we would have

$$\langle \{\, (a, b, c), (a, b, c) \,\}, 5 \rangle.$$

As can be seen, these representations are equivalent.

Since both streams and relations can be seen as streams, we can reclassify the mappings between relations and streams. The relation-to-relation mappings can be thought of as local operators, i.e. they perform some operation on a tuple regardless of its timestamp. The relation-to-stream and stream-to-relation mappings can be thought of as non-local operators, i.e. they are not local operators, therefore their results will change with time.

These concepts require further thought and formalisation.

Chapter 3

R-GMA: An Information Integration System for Grid Monitoring

The work we propose to do will build upon the Relational Grid Monitoring Architecture (R-GMA) system [CGM+03] being developed within the European DataGrid project [Dat03]. We will now give a description of the R-GMA system. In chapter 5 we will describe how we will extend the current R-GMA system.

3.1 R-GMA as a Virtual Data Warehouse

R-GMA is a Grid monitoring system based on the Grid Monitoring Architecture (GMA) introduced in section 2.1.2. It extends the basic architecture by using the relational model to provide a virtual warehouse through which data flows.

R-GMA is not the only proposed system to use the relational model, the Unified Relational Grid Information System [DP01] proposes to use a database management system (DBMS) to store and manage the monitoring information. However, this approach will have a central point of failure and bottleneck, namely the DBMS. R-GMA gets round this by only providing a virtual database.

The relational model provides R-GMA with

- a naming mechanism: the schema which is defined globally for the system, and
- a query language: a subset of SQL.

R-GMA extends the GMA by providing agents who act on behalf of the producers, consumers, and registry[1]. The agents provide the clients of the system with R-GMA specific knowledge that cannot be assumed. The agents are currently implemented as a set of APIs and are hosted on web servers.

[1]The registry is R-GMA's version of the GMA directory service. Further details of the registry are given in section 3.5.

3.2 Querying R-GMA and Consumers

As stated in section 2.1.2, Grid monitoring systems must be able to cope with both static and streaming data. R-GMA provides several modes of querying for streaming data. These are:

Continuous Queries: These are queries that will return tuples which meet the conditions of the query from the point when the query was posed until the query is stopped. The tuples are streamed back to the consumer from the various data sources.

Latest-State Queries: These queries correspond to a window specification using a tuple-based approach with a size of one tuple. That is, it will return the last tuple for each of the key values if this tuple meets the condition of the query.

History Queries: These queries are posed over the elements that have already appeared in the data stream.

It also provides one method for querying static data. These are known as *one-time* queries and correspond to a query over a database maintained in a database management system. However, in reality this could be a data file with a wrapper giving the impression of a relational database.

When a consumer is created in R-GMA, it must register its query with the registry. The consumer can be constructed with any of the above modes of query, depending upon the type of data that the user is interested in. The registry will then inform the consumer of which data sources to contact to retrieve answers for its query.

3.3 Producers

There are two types of primary producer in R-GMA[2]. These are

Stream Producers: Publishes a stream of data, e.g. the memory usage of a computing element. It is capable of answering both continuous and latest-state queries, as it maintains a pool containing the last tuples seen for each value of the primary key.

Database Producers: Publishes a relational database storing more static information about the Grid, e.g. the operating system of a computing element. It is capable of answering one-time queries.

The producers must register a view definition describing the tuples that they make available to the system. R-GMA requires this to be a *sound* description, i.e. all tuples generated by the producer are consistent with the view definition.

3.4 Republishers

These are R-GMA's version of the intermediary introduced by the GMA, section 2.1.2. A republisher takes data from either producers or other republishers and makes the combined answer available for other consumers, i.e. it combines a consumer and a producer in one

[2]Ideally R-GMA would have two producers. There are currently about seven, each with a different mode of querying. All data in R-GMA is currently streamed.

component. The query of the consumer in the republisher is the same as the view definition of the producer in the republisher.

The aim is to have two different types of republishers in R-GMA.

Stream Republishers: Produces a stream of data by merging streams consumed from either stream producers or stream republishers. It is capable of answering both continuous and latest-state queries.

Stream Archivers: This republisher takes a stream as input and stores it in a database. It is then able to answer history queries over the stored stream, e.g. What was the memory usage of computing element X last Thursday?.

We will use the term *publishers* to refer collectively to producers and republishers.

3.5 The Registry

The R-GMA registry plays the role of a GMA directory service. The registry is responsible for matching consumer queries with suitable publishers of information. This is the problem of answering queries using views. However, in R-GMA currently, all the view definitions of the stream publishers are restricted to select queries, i.e. there are no projections or joins. Therefore, to find the maximally contained rewritings does not require the complexity of the bucket algorithm. The R-GMA registry need only return the views which publish for the relation which the consumer query is interested in and that meet the conditions of the query.

The query plans currently returned by the registry will contain redundancy for continuous queries due to presence of republishers. Techniques for providing duplicate free query plans, in the presence of republishers were presented in [CGM+03] and are outlined in chapter 5.

The R-GMA registry is replicated using the gossip architecture to prevent it from being a single point of failure. Each instance of the registry communicates, using update messages, with all the other registry instances to maintain a global view of the system. These update messages are exchanged frequently, so that each registry instance has knowledge of the entire Grid.

3.6 The Schema

One further element, required by R-GMA being an information integration system, is a globally defined schema. This is stored in a database to allow the registry to ensure that queries, and view definitions are well formed, i.e. refer to the correct type of attribute within relations in the global schema.

The schema consists of a set of core relations known as the GLUE schema [Glu03]. These relations were defined by a number of Grid projects, including DataGrid, and describe the components of a Grid, such as computing elements. However, publishers may also introduce new relations into the global schema.

The core relation are split into two sets; static, and stream. The stream relations have an additional attribute that is not defined in the schema, the timestamp which is of type DateTime.

The schema component is also replicated in the R-GMA system. It uses a master/slave mechanism as it is important to ensure that two publishers don't try and introduce the same relation name with different attributes. Updates to the schema are infrequent.

Chapter 4

Proposed Work

We will now outline the problem of managing views over data streams. We will be developing this work in the R-GMA system, but will indicate why it is interesting in general.

4.1 Problem

We aim to provide mechanisms to create, maintain automatically, and query views over distributed data streams. Since we are dealing with data streams, we will need to continually query the data sources in order to maintain the view with fresh data. This will be co-ordinated by a mediator generating suitable query plans for these views over the data streams to ensure that they are sound, complete, and duplicate free.

The benefit of setting up these view mechanisms is for query answering, see example *1*. Also, as the view mechanism provides data from multiple sources at one point, it will allow the distributed system to scale to larger numbers of sources. For the DataGrid setting there are potentially hundreds of thousands of producers each producing a trickle of data. The idea of the republisher is to collect these trickles together to produce streams.

Example 1 Take the R-GMA *situation of computing elements. The global relation*

$$\mathsf{ComputingElement}(\mathsf{machineID}, \mathsf{siteID}, \mathsf{numberOfJobs}, \mathsf{memoryUsage})$$

is used to record the number of jobs running, and the amount of memory being used, on a computing element. The primary key is made up of machineID *and* siteID*, which identify a machine on a site. The attribute* siteID *is a foreign key to some relation maintaining information about the sites on the Grid.*

Each computing element would have a stream producer registered for it with a view definition such as,

```
SELECT *
FROM   ComputingElement
WHERE  machineID = 'linux103' AND
       siteID = 'HW'
```

A resource broker which is trying to locate lightly loaded computing elements, i.e. those with a small number of jobs and low memory usage, would need to retrieve information about every computing element in the system. In order to query every computing element, it would need to pose a query to each computing element to retrieve the required information.

If there were a view maintained with the query,

```
SELECT *
FROM   ComputingElement
WHERE  siteID = 'HW'
```

which collected information about all the computing elements located at one site, in this case Heriot-Watt. The resource broker would then only need to contact the views maintained at each site, rather than each machine at each site.

The idea of setting up and maintaining suitable views presented in example *1* can be extended. We could setup another view which made use of the information collected at the site level views. For example, we could have a view for all computing elements with a memory usage less than 50% which used the site level views to retrieve its information. We call this idea of maintaining views, which draw upon other views for their data, a *hierarchy* of views.

In R-GMA, a stream republisher is essentially a view over the distributed source data streams. The stream republisher hierarchy is a set of views which draw upon other stream publishers. The stream republishers merge streams together and filter according to their view definition. They then publish a derived stream, which it maintains by continually consuming from its own set of publishers.

4.2 Aims

In order to maintain a hierarchy of views (republishers) over distributed data streams (stream producers), we have identified the following aims and objectives:

- Query Planning: the mediator needs to construct suitable query plans for each consumer (including the consumers attached to a republisher), making use of suitable publishers. The load of answering queries should be spread over the entire set of publishers so as not to overload the top level (most general) republishers as this would result in a performance bottleneck.
- Query Execution: Devise suitable protocols for
  - executing query plans;
  - changing between query plans, for example when a new stream republisher is added.

  The protocols need to ensure that consumers get a *sound*, *complete*, and *duplicate free* answer.
- Query Optimisation: Identify the "best" intermediary republishers to create. We will need to define what "best" means. What are the intermediary results required to answer queries? For example, for join queries or for aggregate queries. What views will provide performance gains to the system as a whole?
- Query Complexity: Allow more complex views to be defined, i.e. a select-join view. We will need to define what it means to join a stream of data, both with another stream and with a static source.

In completing this work we will have developed techniques for maintaining a distributed set of views over a set of distributed data streams. This could then be used in the implementation of a distributed data stream management system. We will also have added functionality to the R-GMA system that is to be used in the European DataGrid project.

Chapter 5

Current and Future Work

We will discuss how we intend to conduct our research. We will then present ideas that have been developed, between Werner Nutt, Andy Cooke, Lisha Ma, and myself, on which we will build. Finally, we present some ideas that I developed to enable R-GMA to handle queries containing comparisons.

5.1 Approach

We will follow an iterative approach to realise the aims and objectives identified in section 4.2. We will begin with simple continuous conjunctive select queries, e.g.

```
SELECT *
FROM   computingElement
WHERE  siteID = 'HW' AND
       numberOfJobs <= 5;
```

We will then increase the complexity of the queries to allow for joins on latest-state information, continuous join queries, aggregate queries, "mixed joins" (i.e. joining static and streaming data). For each type of query we will consider the query planning required, how to execute the query and how this is affected by the addition or removal of publishers. We will also consider the infrastructure, e.g. the views to be maintained, that would benefit the execution of such queries, e.g. suitable intermediary results for join queries.

5.2 Query Planning

Each of the components in R-GMA has been implemented as an API in several programming languages. The registry functionality has been split into several classes. The registry public API acts as an interface to the rest of the classes in the registry. The classes which perform the work of the registry are:

Registrar: responsible for maintaining the set of publishers and consumers known to the system.

Mediator: responsible for matching consumers with relevant publishers.

We will begin our work by extending the functionality of the mediator to allow the mediator to generate a query plan bank. A query plan bank contains a set of equivalence classes. The consumer generates a query plan by selecting one publisher from each of the classes in the query plan bank.

We will now define the properties that we would like a query plan to maintain and discuss how a query plan bank can be generated. The following subsections, upto section 5.2.4, come from [CGM+03] and were also presented in [NCGM03].

5.2.1 Properties of Streams and Stream Publishers

For simplicity we assume that every stream publisher publishes for a single relation. We can then identify a publisher by its relation. For each publisher P the registry will store the descriptive view

$$\sigma_D(r),$$

which is a selection on some global relation r.

We require tuples to follow a route defined by the key attributes, called a *channel*. A stream S should have the following properties,

Duplicate Free Stream it never outputs the same tuple twice.

Weakly Ordered Stream the tuples belonging to one channel appear in the order of their timestamps.

Sound w.r.t. a view all the tuples in the stream satisfy the condition of the view.

Complete w.r.t. a view it contains every tuple in the system that satisfies the view.

Two streams are *disjoint* if they do not share any tuples. A publisher is duplicate free, weakly ordered, sound w.r.t. a view, or complete w.r.t. a view if its stream is. Likewise, two publishers are disjoint if their streams are disjoint.

R-GMA uses the assumption that all producers are disjoint. This cannot not be enforced due to the distributed nature of the system. Although it could be checked by ensuring that the new view is disjoint from all other views currently registered. However, it is likely to hold as many stream producers will be registering views to publish sensor data, which by the design of the global schema will make the definitions disjoint.

Proposition 2 *If all stream producers are sound w.r.t. their views, then all stream producers are,*

- *mutually disjoint;*
- *complete w.r.t. their views.*

5.2.2 Query Plans and their Properties

A *plan* for a query $q = \sigma_C(r)$ is an expression

$$\sigma_{C_1}(P_1) \uplus \ldots \uplus \sigma_{C_m}(P_m), \tag{5.1}$$

where $P_1, \ldots, P_m$ are stream publishers, each described by a view $\sigma_{D_i}(r)$ and C_i is the condition of the query posed at publisher P_i.

A query plan may give rise to duplicates as it potentially contains republishers which overlap. Query plans should have the following properties,

Sound w.r.t. a query Assuming that the publishers in the system are sound, a query plan (5.1) is *sound* w.r.t. a query if and only if the conditions of the query posed at the publisher C_i and the conditions of the view of the publisher D_i together entail the condition of the query C

$$C_i \wedge D_i \models C.$$

Complete w.r.t. a query Assuming that the publishers in the system are complete, a query plan (5.1) is *complete* w.r.t. a query if and only if the condition of every producer with answers for the query are entailed by some condition in the query plan

$$C \wedge E \models \bigvee_{i=1}^{m} C_i \wedge D_i,$$

where E is the condition of the stream producer.

Duplicate Free Assuming that the publishers in the system are duplicate free and sound w.r.t. their views, a query plan (5.1) is *duplicate free* if and only if the conditions posed to each publisher in the plan do not overlap

$$(C_i \wedge D_i) \wedge (C_j \wedge D_j),$$

are unsatisfiable for all $i, j \in 1 \ldots m$ with $i \neq j$.

To ensure that the derived stream is weakly ordered requires all the value attributes for one key to all follow the same channel. We call this being *faithful*. We represent the set of key values as x and their corresponding value attributes with the set y. A query plan is said to be *faithful* if all the values y for a channel x come from the same publisher P_i.

To simplify our presentation we will use $F_i(x, y)$ to represent $C_i(x, y) \wedge D_i(x, y)$. A query plan can be checked to be faithful if proposition 3 holds for the plan.

Proposition 3 *Suppose* $C(x, y) = C^k(x) \wedge C^v(y)$ *and* $F_i(x, y) = F_i^k(x) \wedge F_i^v(y)$. *Then*

$$\exists y.(C(x, y) \wedge F_i(x, y)) \models \forall y.(C(x, y) \implies F_i(x, y))$$

holds iff one of the following holds:

1. $C^k(x) \wedge F_i^k(x)$ *is unsatisfiable;*
2. $C^v(y) \wedge F_i^v(y)$ *is unsatisfiable;*
3. $C^v(y) \models F_i^v(y)$.

5.2.3 Constructing Query Plans

Query plans are constructed to be sound and complete w.r.t. the query, duplicate free, and faithful. This is achieved in the way the plans are constructed. However, we first need to identify which publishers can provide answers to the user query.

Definition 11 (Relevant Publishers) *A publisher with view definition* $v = \sigma_{D^k \wedge D^v}(r)$ *is* relevant *for query* $q = \sigma_{C^k \wedge C^v}(r)$ *if for some channel it delivers all values, i.e.*

1. $C^k \wedge D^k$ *is satisfiable;*
2. $C^v \models D^v$.

The set of relevant publishers in R-GMA can be constructed using an SQL query. Further details of this are given in section 5.4.

A publisher P_1 is said to be more general than another publisher P_2 if P_1 is *subsumed* by P_2 (definition 12). If P_1 is subsumed by P_2, but P_2 is not subsumed P_1, then we say that P_1 is *strictly* subsumed by P_2, written $P_1 \prec P_2$.

Definition 12 (Subsumption of Publishers) *Consider publishers* P_1 *with view definition* $\sigma_{D_1}(r)$ *and* P_2 *with view definition* $\sigma_{D_2}(r)$*, and the query* $q = \sigma_C(r)$*. Then*

- P_1 *is* subsumed *by* P_2 *w.r.t. q, written* $P_1 \preceq_q P_2$*, if*

$$C \wedge D_1 \models D_2.$$

- P_1 *is* subsumed *by* P_2*, written* $P_1 \preceq P_2$*, if*

$$D_1 \models D_2.$$

The construction of query plans is done in two stages within R-GMA. The first is performed by the registry and its components, i.e. the mediator. The registry constructs a set of relevant publishers for the query. From this, the set of *maximally relevant publishers* for the query is generated. A publisher is maximally relevant for the query if there are no publishers which strictly subsume it w.r.t. the query.

Due to the redundancy in the system, introduced by the republishers, there might be several alternative maximally relevant publishers to generate the set of answers. The registry passes the consumer agent all possible plans as a query plan bank. The publishers in the query bank are organised into equivalence classes based on the answers that they can provide the query and then topologically sorted by their subsumption of each other in general.

The consumer agent is then responsible for constructing the actual query plan for execution from the query plan bank. It takes one publisher from each equivalence class in the query plan bank and contacts it. If the publisher cannot be contacted for some reason, then the next publisher in that equivalence class is contacted.

When the consumer agent constructs the plan from the query plan bank, it must construct a *canonical plan*. This is a plan that uses one publisher from each of the equivalence classes and has the form

$$\sigma_{C_1}(P_1) \uplus \ldots \uplus \sigma_{C_m}(P_m),$$

where $C_1 = C$ and each subsequent $C_i = C \wedge \neg D_1 \wedge \ldots \wedge \neg D_{i-1}$. Constructing the conditions of the query like this ensures that we get no duplication of tuples in the answer set, and that all tuples for a given key value all follow the same channel through the system.

5.2.4 Design and Implementation

We will implement the logic outlined above in the R-GMA mediator component. We will do this by adding additional classes to represent,

- Contributors: these are all the publishers that can potentially offer tuples to the query. This class will offer two methods:
  1. test whether the condition of the contributor is subsumed by another contributor.
  2. test whether the condition of the contributor is subsumed w.r.t. the query by another contributor.
- A bank of query plans: these will be passed to the consumer to allow it to construct an instance of the plan. The bank contains all the contributors grouped into equivalence classes based on the tuples they can offer a query.

Our proposal for the redesign of the mediator class by the addition of two classes is given as a class diagram in figure 5.1. This class diagram also shows the interaction of the various classes of R-GMA at present.

Our work will also re-implement the stream producer and stream republisher so as to be able to maintain a pool of latest-state values.

5.3 Query Execution

Once the canonical query plans have been generated, giving the query that will need to be posed to each publisher in the plan, they will be executed by the consumer agent.

The problem we will try to solve is when a consumer's query plan, or a republisher's consumer's query plan, needs to be updated, for instance if a republisher in the query plan is removed from the system. The plan can easily be updated by selecting another publisher from the same equivalence class in the query plan bank. However, we must ensure that during the transition between the two query plans the consumer receives the full set of tuples without duplicates. We will now provide details of the different situations for the protocol.

5.3.1 Adding a New Stream Producer

The new producer might be publishing data that is relevant to existing consumers, including republishers. The registry must ensure that every consumer for which the new producer publishes data receives all that data. However, not all consumers will need to be informed as they can receive the tuples via a republisher it already consumes from.

5.3.2 Removing a Stream Producer

This is the simplest case. If the stream producer S ceases to exist then any consumer having S in their query plan bank need to be informed to remove S from their query plan bank.

5.3.3 Adding a New Stream Republisher

When a new stream republisher is added to the system then it might be possible to improve the query plans of consumers for which it is relevant. The new republisher will only improve

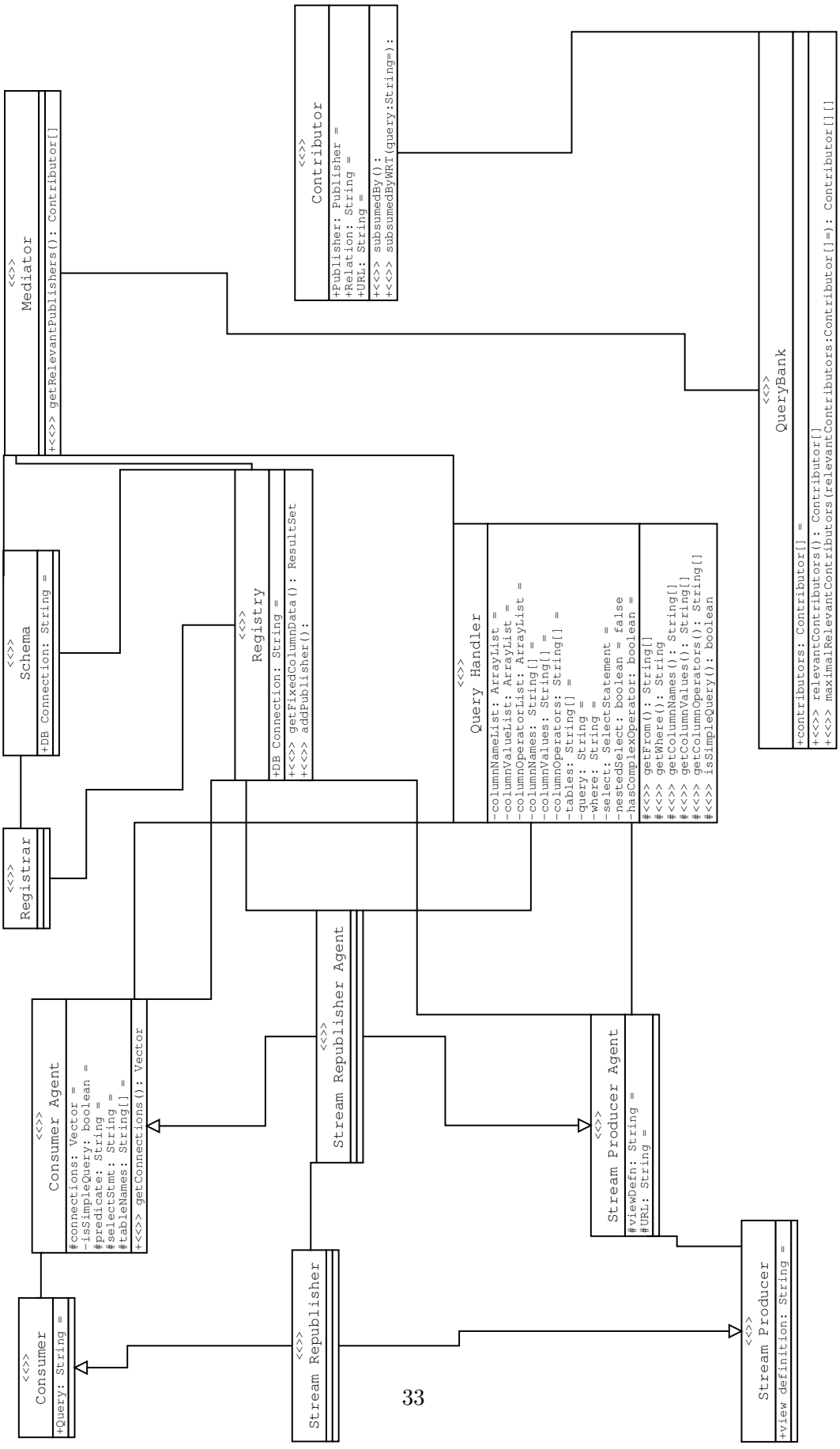

Figure 5.1: Class Diagram of the R-GMA System

query plans when it subsumes w.r.t. the query one or more publishers in the consumer's query plan. However, care must be taken to ensure that the new republisher has become complete w.r.t. its view definition before it is used by other consumers, i.e. the republisher has contacted and begun streaming from all of the publishers in its query plan.

5.3.4 Removing a Stream Republisher

When a stream republisher is to be removed from the system, all the consumers who have the republisher in their query plan bank must be informed to remove it. In the case that the republisher was the only publisher in its equivalence class, the registry will need to supply a patch. The naive approach would be to supply all the publishers who the republisher consumes from. However, some of these may be subsumed by other publishers in the query plan bank.

5.4 Comparison Queries

Part of the work that I have undertaken during the first year is to consider more complex continuous queries and view definitions. The current R-GMA restricts continuous queries and view definitions to conjunctive select queries with only `attribute = value` comparisons. We have generated a suitable SQL query for generating the set of relevant publishers when the continuous queries and view definitions may contain comparison operators from $\{<, \leq, =, \geq, >\}$.

The SQL query is posed at a database storing the publishers known to the system with their fixed values. For example, say we had a stream producer for the computing element linux103 at Heriot Watt. Based on the schema presented in example *1*, the values for the machineID and siteID would be fixed to linux103 and HW respectfully.

The SQL query first proceeds by finding all publishers for the relation that the user query is interested in. Then, based on the user query, we identify the publishers whose view condition would *contradict* the conditions of the query and thus be of no use in providing answers. The cases when a view definition contradicts a query's conditions are presented in table 5.1. The conditions of the query and the view are represented as intervals of the form

$$]a, b],$$

which is interpreted as the value between a and b, not including the value a.

To generate the SQL query, a case analysis needs to be performed on the user query in some external code. Then, a suitable nested query can be posed at the registry's database where publisher's view definitions are stored.

View Condition

	$[a,b]$	$]a,b]$	$[a,b[$	$]a,b[$	$]-\infty,b]$	$]-\infty,b[$	$[a,\infty[$	$]a,\infty[$	$]-\infty,\infty[$
$[c,d]$	$d < a \vee b < c$	$d \leq a \vee b < c$	$d < a \vee b \leq c$	$d \leq a \vee b \leq c$	$b < c$	$b \leq c$	$d < a$	$d \leq a$	
$]c,d]$	$d < a \vee b \leq c$	$d \leq a \vee b \leq c$	$d < a \vee b \leq c$	$d \leq a \vee b \leq c$	$b \leq c$	$b \leq c$	$d < a$	$d \leq a$	
$[c,d[$	$d \leq a \vee b < c$	$d \leq a \vee b < c$	$d \leq a \vee b \leq c$	$d \leq a \vee b \leq c$	$b < c$	$b \leq c$	$d \leq a$	$d \leq a$	
$]c,d[$	$d \leq a \vee b \leq c$	$d \leq a \vee b \leq c$	$d \leq a \vee b \leq c$	$d \leq a \vee b \leq c$	$b \leq c$	$b \leq c$	$d \leq a$	$d \leq a$	
$]-\infty,d]$	$d < a$	$d \leq a$	$d < a$	$d \leq a$			$d < a$	$d \leq a$	
$]-\infty,d[$	$d \leq a$	$d \leq a$	$d \leq a$	$d \leq a$			$d \leq a$	$d \leq a$	
$[c,\infty[$	$b < c$	$b < c$	$b \leq c$	$b \leq c$	$b < c$	$b \leq c$			
$]c,\infty[$	$b \leq c$	$b \leq c$	$b \leq c$	$b \leq c$	$b \leq c$	$b \leq c$			
$]-\infty,\infty[$									

Table 5.1: Cases when a query interval a, b contradicts a view's interval c, d

Bibliography

[ABW03] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. In *Proceedings of the Ninth International Conference on Data Base Programming Languages*, September 2003. Short version of a technical report. Full version can be found at http://www-db.stanford.edu/∼widom/pubs.html#streams.

[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, June 1-3, 1998, Seattle, Washington*, pages 254–263. ACM Press, 1998.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 487–498, May 2000.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 275–286, June 1999.

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 1996 Annual ACM Symposium on Theory of Computing*, pages 20–29, 1996.

[BBD+02] B. Babcock, Sh. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *proceedings 21st Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM, 2002.

[BCC+02] Rob Byrom, Brian Coghlan, Andrew Cooke, Roney Cordenonsi, Linda Cornwall, Ari Datta, Abdeslem Djaoui, Laurence Field, Steve Fisher, Steve Hicks, Stuart Kenny, James Magowan, Werner Nutt, David O'Callaghan, Manfred Oevers, Norbert Podhorszki, John Ryan Manish Soni, Paul Taylor, Antony Wilson, and Xiaomei Zhu. R-gma: A relational grid information and monitoring system. In *2nd Cracow Grid Workshop*, Poland, December 2002.

[BCC+03] Rob Byrom, Brian Coghlan, Andrew Cooke, Roney Cordenonsi, Linda Cornwall, Ari Datta, Abdeslem Djaoui, Laurence Field, Steve Fisher, Steve Hicks, Stuart Kenny, James Magowan, Werner Nutt, David O'Callaghan, Manfred Oevers, Norbert Podhorszki, John Ryan Manish Soni, Paul Taylor, Antony Wilson, and Xiaomei Zhu. R-gma: First results after deployment. In *Computing in High Energy and Nuclear Physics*, La Jolla, Ca, USA, March 2003.

[Ber01] Francine Berman. From TeraGrid to Knowledge Grid. *Communications of the ACM*, 44(11):27–28, 2001.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *proceedings 28th International Conference on Very Large Data Bases*, August 2002.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *proceedings 10th Symposium on High Performance Distributed Computing*, 2001.

[CGM+03] Andy Cooke, Alasdair J. G. Gray, Lisha Ma, Werner Nutt, James Magowan, Paul Taylor, Rob Byrom, Laurence Field, Steve Hicks, Jason Leake, Manish Soni, Anthony Wilson, Roney Cordenonsi, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian Coghlan, Stuart Kenny, and David O'Callaghan. R-GMA: An information integration system for grid monitoring. In *11th International Conference on Cooperative Information Systems, Sicily, Italy*, Lecture Notes in Computer Science, New York, NY, USA, November 2003. Springer-Verlag Inc.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM Press, 1977.

[CMN99] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 263–274, June 1999.

[CNS01] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among aggregate queries with negation. In *Proceedings 20th Symposium on Principles of Database Systems*, Santa Barabara (Ca), May 2001.

[Dat03] The datagrid project. http://www.eu-datagrid.org, June 2003.

[DP01] P. Dinda and B. Plale. A unified relational approach to grid information services. In *GIS-WG Program at the GGF1*, number GWD-GIS-012. Global Grid Forum, March 2001.

[FK99] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2: Computational Grids. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[FM83] P. Flajolet and G. Martin. Probabilistic counting. In *In Proceedings of the 1983 Annual IEEE Symposium on Foundations of Computer Science*, 1983.

[GGF03] Global grid forum. http://www.ggf.org, 2003.

[Glo03] Globus toolkit. http://www.globus.org, June 2003.

[Glu03] High energy nuclear physics intergrid collaboration board. http://hicb.org/, June 2003.

[GM99] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Database Theory - ICDT'99: 7th International Conference*, volume 1540 / 1998, page 332, Jerusalem, Israel, January 1999. Springer-Verlag Heidelberg.

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

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proceedings of the 1999 International Conference On Very Large Data Bases*, pages 174–185, September 1999.

[JGN99] W.E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: the engineering aspects of nasa's information power grid. In *proceedings 8th IEEE International Symposium on High Performance Distributed Computing*, pages 197–204. IEEE, 1999.

[Klu88] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[Lev00] Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 575–595. Kluwer Academic Publishers, Dordrecht, 2000.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the twenty-second international Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 251–262, Los Altos, CA 94022, USA, 1996. Morgan Kaufmann Publishers.

[Man01] Robert C Mann. Astrogrid: the uk's virtual observatory initative. In *Eleventh Astronomical Data Analysis Software and Systems*, Vicotria, BC, Canada, October 2001.

[NCGM03] Werner Nutt, Andy Cooke, Alasdair J G Gray, and Lisha Ma. Republisher hierachies in r-gma. Seminar Presentation, July 2003.

[PG99] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *Proceedings of the 1999 International Conference on Scientific and Statistical Database Management*, pages 24–33, July 1999.

[PH01] Rachel Pottinger and Alon Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal – The International Journal on Very Large Data Bases*, 10(2-3):182–198, 2001.

[RGM03] DataGrid WP3 Information and Monitoring Services. http://hepunx.rl.ac.uk/edg/wp3/, June 2003.

[RVSR98] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: adaptive control of distributed applications. In *Seventh Int'l Symp. High Performance Distributed Computing*, 1998.

[Sha03] Cyrus Shahabi. Aims: an immersidata management system. In *CIDR 2003*, Asilomar, CA, January 5–8 2003.

[Smi02] Warren Smith. A system for monitoring and management of computational grids. In *proceedings 2002 International Conference on Parallel Processing*, 2002.

[Sul96] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *proceedings 22nd International Conference on Very Large Data Bases*, page 594, September 1996.

[TAG+00] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture. Global Grid Forum Performance Working Group, March 2000. Revised January 2002.

[Ull97] J.D. Ullman. Information integration using logical views. In F.N. Afrati and Ph.G. Kolaitis, editors, *proceedings International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, January 1997.

[vdM92] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California*, pages 331–345. ACM Press, 1992.