Chapter 1

Data Integration Services

1 Introduction

Data integration systems harmonize data from multiple sources into a single coherent representation. The goal is to provide an integrated view over all the data sources of interest and to provide a uniform interface to access all of these data. The access to the integrated data is usually in the form of querying rather than updating the data.

1.1 Motivation for Data Integration Systems

- Users can focus on specifying *what* they want, not on *how* to obtain what they want. Instead of finding relevant sources, interacting with every source and combining data from different sources, a user can ask queries in a unified way.
  Particular examples:

  - Desire for reports that describe all parts of a merged organization (bank mergers, car dealerships, etc.).

- Facilitates decision support applications (OLAP, Data mining)

  OLAP (On-Line Analytical Processing) is making financial, marketing or business analysis to be able to make business decisions on a collection of detailed data from one or more data sources. The analysis is done through asking large number of aggregate queries on the detailed data.

  Data Mining is discovering knowledge from a large volume of data. Statistical rules or patterns are automatically found from the raw collection of data.

1.2 Major Issues

The data sources to be integrated may belong to the same enterprise, but might also be arbitrary sources on the World Wide Web. Most of the time, each of the sources is independently designed for autonomous operation. Also, the sources are not necessarily databases; they may be legacy systems (old and obsolescent systems that are difficult to migrate to a modern technology) or structured/unstructured files with different interfaces. Data integration requires that the differences in modeling, semantics and capabilities of the sources together with the possible inconsistencies be resolved. More specifically, the issues that make integrating such data difficult include:

- Differences in how data sources model the world
- Differences in how data sources represent data
- Availability of data sources
- Autonomy of data sources - the fact that their data and/or schema can change without notice
- Conflicting data from two or more data sources
- Correctness of the integrated view of the data
- Query performance

This chapter investigates many of the problems that occur in data integration systems, and some of the solutions designed to deal with the problems. We outlined the chapter as follows: First, we present three architectural approaches to data integration systems in Section 2. Section 3 discusses semantic problems that arise when multiple heterogeneous sources are integrated. Techniques for querying the integrated data are presented in Section 4. The data extraction phase of querying where data is actually obtained from the data sources is detailed in Section 5. Management of materialized views in datawarehousing systems is provided in Section 6. Later we present some example systems in the following section. Finally, we conclude the chapter after a discussion of research issues and open problems in Section 8.

2 Data Integration Architectures

2.1 Dimensions to Categorize Architectural Models for Integrating Data Sources

There are three orthogonal dimensions which are traditionally used in literature to describe distributed information systems: autonomy, heterogeneity and distribution. Sometimes transparency is considered as the forth parameter. Below we are discussing each of these dimensions.

- Autonomy
  Autonomy refers to the degree to which individual data sources can operate independently. According to Veijalainen and Popescu-Zeletin's classification [MW88] there are three types of autonomy:
  - Design autonomy
    The source is independent in data models, naming of the data elements, semantic interpretation of the data, constraints etc.
  - Communication autonomy
    The source is independent in deciding what information it provides to the other components that are part of the integrated system and to which requests it responds.
  - Execution autonomy
    The source is independent in execution and scheduling of incoming requests.
- Heterogeneity
  Heterogeneity refers to the degree of dissimilarity between the component data sources that make up the data integration system. It occurs at different levels. On a technical level, heterogeneity comes from different hardware platforms, operating systems, networking protocols, and similar lower-level concepts. On a conceptual level, heterogeneity comes from different programming and data models as well as different understanding and modeling of the same real-world concepts (*i. e.*, naming of relations and attributes).
- Distribution
  Distribution refers to the physical distribution of data over multiple sites.
  - Client/Server
    Server does data management, client provides user interface.
  - Peer-to-Peer (fully distributed)
    Each machine has full functionality of data management.
- Transparency
  Transparency refers to the separation of higher-level semantics of a system from lower-level implementation issues. A transparent system hides the implementation details from users.

2.2 Major Approaches to Data Integration

Three common approaches to integrate data sources are the following:

- Virtual View Approach
  In this case data is accessed from the sources on-demand (when a user submits a query to the information system). This is called a *lazy approach* to data integration.

- Materialized View/Warehousing Approach
  Some filtered information from data sources is pre-stored in a repository (warehouse) and can be queried later by users. This method is called an *eager approach* to integration.

- Hybrid approach
  Integrated data is selectively materialized. The system is essentially mediator-based where data is extracted from sources on-demand, but the results of some queries are pre-computed and stored. In order to choose what queries to materialize, designers should consider many factors, such as "popularity" of queries and cost of maintenance [Ash00].

When the number of data sources in an integrated system is very large, and/or the sources are prone to change often (like in the case of web sources), and/or there is no way to predict what kind of queries users will ask, the virtual view approach is preferable to the data warehousing approach. If, however, sources are fixed, don't get upgraded too often and designers of the integrated system know what kind of queries are most popular, we can materialize some of them.

2.3 Virtual View Approach

2.3.1 Federated Database Systems

A *Federated Database System (FDBS)* consists of semi-autonomous components (database systems) that operate independently but participate in a federation to partially share data with each other [SL90].

This sharing is controlled by each component and is not centralized. The components can not be called "fully-autonomous" because each component is modified by adding an interface that allows communication with all other databases in the federation.

Each of the component database systems can be either a centralized DBMS, a distributed DBMS, or another federated database management system, and may have any of the three types of autonomy mentioned above (design autonomy, communication or execution autonomy). As a consequence of this autonomy, heterogeneity issues become the main problem.

A FDBS supports local and global operations, and treats them differently. Local operations involve only local data access and correspond to the queries submitted directly to this source. Global operations use a *FDMS (Federated Database Management System)* to access data from other components. In case of a global operation, each data source whose data is required must allow access to that data.

There are *loosely coupled* FDBSs and *tightly coupled* FDBSs. A tightly coupled FDBS has a unified schema (or several unified schemas) which can be either semi-automatically built by schema integration techniques (see Section 3 for details) or created manually by user. To solve logical heterogeneity, a domain expert needs to determine correspondences between schemas of the sources. Tightly coupled FDBS is usually static and difficult to evolve, because schema integration techniques don't allow to add or remove components easily. Examples of this kind of FDBSs are ...

A loosely coupled FDBS does not have a unified schema, but it provides some unified language for querying sources. In this configuration, component database systems have more autonomy, but humans must resolve all semantic heterogeneities. Only technical metadata is needed by loosely coupled FDBS as opposed to tightly coupled one, which requires semantic metadata in addition. Requested data comes from the exporter of this data itself and each component can decide how it will view all the accessible data in the federation. As there is no global schema, each source can create its own "federated schema" for its needs. Examples of such systems are MRSDM [Lit85], Omnibase [Rea89] and Calida [JPSL$^+$88].

As pointed out by D. Heimbigner and D. McLeod [HM85], in order to remain autonomously functioning systems and provide mutually beneficent sharing of data at the same time, components of FDBS should have facilities to communicate in three ways:

- Data exchange
  This is the most important purpose of the federation and good mechanisms of data exchange are a must.

- Transaction sharing
  There may be cases where for some reason the component does not want to provide direct access to some of its data, but can share operations on its data. Then other components should have the ability to specify which transactions they want to be performed by another component.

- Cooperative activities
  As there is no centralized control, cooperation is the key in federation. Each source should be able to perform a complex query involving accessing data from other components (?).

The simplest way to achieve interoperability is to map each source's schema to all others' schemas. It is a so-called pair-wise mapping. You can see an example of such federated database system in Figure 1.1. Unfortunately, it requires $n \cdot (n-1)$ schema translations and becomes too tedious with the growth of a number of components in federation. Research is being done on tools for efficient schema translation (See Section 3 for details).

We should note that the term "Federated Database Systems" is used differently in literature: some people call only tightly coupled systems FDBSs, some call only loosely coupled systems FDBSs, and some take the same approach we did by considering tight and loose architectures be two kinds of federated database system architecture.

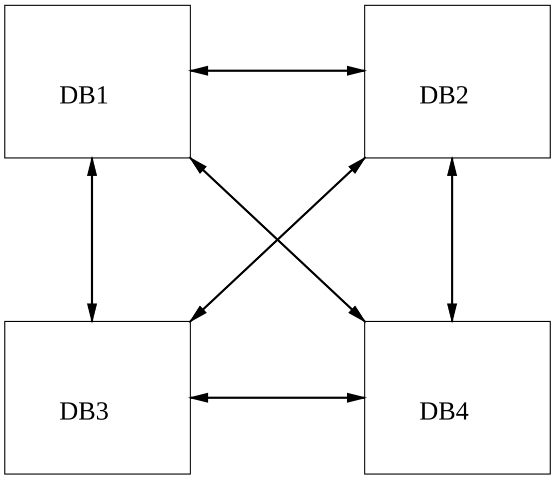

Figure 1.1: Example of federated database architecture (borrowed with some changes from [GMUW00])

Federated architecture is very appropriate to use when there is a number of autonomous sources, and we want, on one hand, to retain their "independence" allowing user to query them separately, and, on the other hand, allow them to collaborate to answer the query. It is a good compromise between full integration and no integration.

2.3.2 Mediated Systems

Mediated system integrates heterogeneous data sources (which can be databases, legacy systems, web sources, etc) by providing virtual view of all this data. Users asking queries to the mediated system do not have to know about data source location, schemas or access methods, because such system presents one global schema to the user (called *mediated schema*) and users ask their queries on it.

A natural question that arises is how mediation architecture is different from tightly coupled FDBS? Here are the differences between them [SL90]:

- A mediated architecture may have non-database components
- The query capabilities of sources in mediator-based system can be restricted and the sources do not have to support SQL-querying at all
- Access to the sources in a mediator-based system is usually read-only as opposed to read-write access in a FDBS

- Development of mediated systems is usually done in top-down way as opposed to bottom-up approach for tightly coupled FDBS
- Sources in a mediator-based approach have complete autonomy which means it is easy to add or remove new data sources

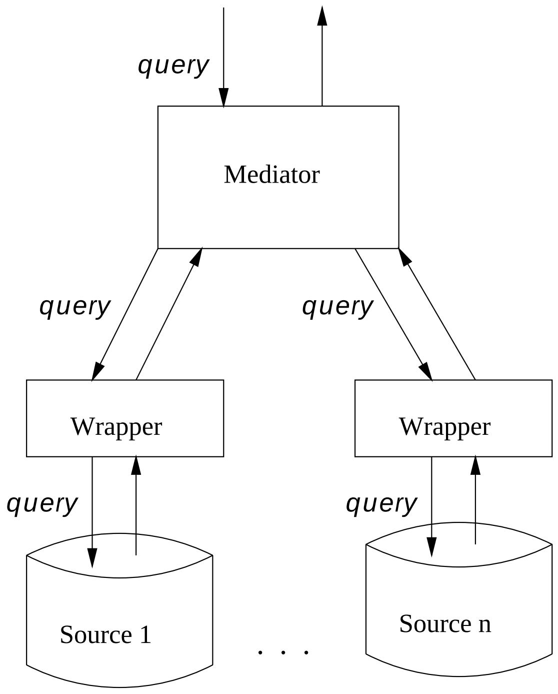

Figure 1.2: Mediated architecture (borrowed with some minor changes from [GMUW00])

A typical architecture for a mediated system (with two sources) is given in Figure 1.2. Two main components of a mediated system are the *mediator* and one *wrapper* per data source. The mediator (sometimes also called an *integrator*) performs the following actions in the system:

- Receives a query formulated on the unified (mediated) schema from a user.
- Decomposes this query into sub-queries to individual sources based on *source descriptions* that the mediator has.
- Optimizes the execution plan based on source descriptions.
- Sends sub-queries to the wrappers of individual sources, which will transform these sub-queries into queries over sources' local models and schemas. Then it receives answers to these sub-queries from wrappers, combines them into one answer and sends it to the user.

These steps are described in detail in Section 4.

A wrapper hides technical and data model details from the mediator. It is an important component of both mediator-based architecture and data warehouse, but wrappers for mediated systems are usually more complicated. Please refer to Section 5.1 for more information about wrappers.

Example: Let us assume there are two data sources - two car dealers databases which both became parts of Acme Cars company. Each of the car dealers has a separate schema for storing information about cars. Dealer one stores it as one relation:

Cars(vin, make, model, color, price, year, mileage)

Dealer two also rents some of his cars, so he has separate relations for cars for sale and cars for rent. He stores information about cars for sale in two relations:

CarsForSale(vechicalID, carMake, carModel, carColor, carPrice, carYear),
CarsSaleMileage(vechicalID, mileage).

Acme Cars uses a mediated architecture to integrate these two dealers' databases. It does it by providing a mediated schema of the two schemas above. The mediated schema consists of just one relation:

Automobiles(vin, autoMake, autoModel, autoColor, autoPrice, autoYear).

Now if a client of Acme Cars submits an SQL-query:

SELECT vin, autoModel, autoColor, autoYear
FROM Automobiles
WHERE autoMake = "Honda" **AND** autoPrice < 14,000

The wrapper for the first database will translate this query to:

SELECT vin, model, color, year **FROM** Cars
WHERE make = "Honda" **AND** price < 14,000

It also renames model to autoModel, color to autoColor and year to autoYear.

The wrapper for the second dealer will translate this query to:

SELECT vechicalID, carModel, carColor, carYear
FROM CarsForSale
WHERE carMake = "Honda" **AND** carPrice < 14,000

The wrapper also renames vechicalID to vin, carModel to autoModel, carColor to autoColor etc.

Some known implementations of mediator-based architecture are: TSIMMIS, Information Manifold, SIMS, Carnot ... Some of them are covered in more details in Section 7.

2.4 Materialized View Approach (Data Warehousing)

In a materialized view approach, data from various sources is integrated by providing a unified view of this data, like in a virtual view approach, but here this filtered data is actually stored in a single repository (called "data warehouse"). How is a data warehouse different from a traditional databases with OLTP (On-Line Transaction Processing)?

- Warehouse usually contains terabytes of data and may combine data from many databases, semi-structured and other sources
- Workloads are query intensive; queries are complex and query throughput is more important than transaction throughput
- A data warehouse often contains historical and summarized data which is used for decision support. That also implies that users of a data warehouse are different than users of a traditional DBMS: they will be analysts, knowledge workers, executives
- Information is usually read-only as opposed to read/write operations in OLTP.

There are three important steps involved in building a data warehouse:

- Modeling and design

  In the stage of designing a warehouse, we need to decide what information from each source we are going to use in the warehouse, what views (queries) over these sources we want to materialize, and what the global unified schema of the warehouse will be.

- Maintenance (refreshing)

  Maintenance deals with how we initially populate our warehouse from source data and how we refresh it when data in the sources are updated. There are three ways to create a warehouse:

  - Do it periodically when no queries to the system are sent (at night, for instance) re-populating a warehouse from scratch from its data sources

  - Periodically *incrementally update* the warehouse, that is, incorporate changes made to the sources since last update. In this case, only very small amount of data will be touched, so it is more efficient. On the other hand, it is more complicated, has a number of issues and is an area of active research.

- Update the warehouse after every change made to any of the sources. This approach does not seem to be very practical, except in small warehouses with rarely changing data sources [GMUW00].

  View maintenance is a key research topic specific to data warehousing and we discuss it in detail in Section 6.

- Operation

  Operation of a data warehouse involves query processing, storage and indexing issues.

Example of a two-source data warehouse is given in Figure 1.3.

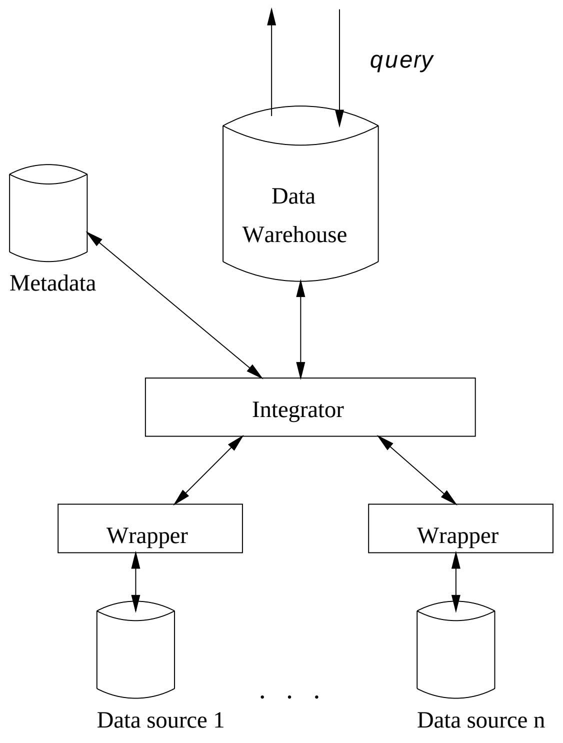

Figure 1.3: A data warehouse

Example: Let's suppose there is a company Cute Toys that owns two toy stores. There are two types of toys at each store: teddy-bears and dogs. Each store has a database, where they store a number toys sold on each date, for each kind of a toy. So store 1 stores relation: **Sales(date, typeToy, numberSold)** and store 2 has two relations: **TeddyBears(date, numberSold)** and **DogsToys(date, numberSold)**.
Now assume, that the company would like to have the following relation in the data warehouse for decision making purposes (future marketing):

ToySales(date, typeToy, numberSold)
In this case, we need to first select appropriate tuples from each source, take their union and then aggregate, so that for each date and type of a toy we have a total number of toys of this kind sold on a given date. The SQL query to the first source is straightforward, as the relation is exactly the same apart from the name it has. It will look like this:

```
INSERT INTO ToySales1(date, typeToy, numberSold)
SELECT date, typeToy, numberSold
FROM Sales
```

For the second source, we can ask two queries:

```
INSERT INTO ToySales2(date, typeToy, numberSold)
SELECT date, "TeddyBear", numberSold
FROM TeddyBears

INSERT INTO ToySales2(date, typeToy, numberSold)
SELECT date, "Dog", numberSold
FROM DogsToys
```

So, wrappers to sources 1 and 2 will return relations ToySales1 and ToySales2 correspondingly. Now integrator component will join them summing the number of toys of each kind sold on each date:

```
INSERT INTO ToySales(date, typeToy, numberSold)
SELECT date, typeToy, SUM(numberSold)
FROM ToySales1 s1, ToySales2 s2
WHERE s1.typeToy=s2.type AND s1.date = s2.date
```

Known implementations of data warehousing approach include Squirrel[] and WHIPS [] systems. An overview of WHIPS is given in Section 7 of this chapter. Two popular applications of data warehousing are OLAP (online analytical processing) and data mining. They are discussed in Chapter **??**.

2.5 Hybrid Approach

This approach is usually discussed as a way to improve performance of some mediator-based systems. Approach to data integration in this case is virtual, but selected queries are materialized in some repository. This repository then can serve as a new source for this mediated system.

Issues which arise in this case are some of issues for data warehousing approach:

- What data to materialize?
- How this materialized data is maintained

A hybrid approach is proposed in [Ash00], but otherwise is less commonly discussed in academic literature than are data warehousing and mediation.

3 Semantic problems in data integration

3.1 Introduction

Different information systems can use different ways of presenting their information to their users. Those differences can make it very difficult for developers to integrate data from the two systems.

The task of integrating information systems is also sometimes referred to as *database integration* [BOT86], *schema integration* [BLN86], *schema merging* [BDKV92], *database integration* [BOT86], or the creation of *multidatabase systems* [Mot99].

This section explores the nature of why this integration can be so difficult, and presents some (partial) solutions the problem.

3.2 The goal of an information system

3.2.1 Information systems as assertions about our world

An information system can be thought of as a record of some facts about the world. (Other functionality might also exist in an information system, but that isn't relevant to this present discussion.)

If an information system is accurate in what it claims about the world, then it's useful. If the information system is inaccurate, the it's much less useful.

3.2.2 System interfaces

An information system makes its information available to users via its *system interfaces*. In modern information systems, these might include any of the following:

- SQL access and the documentation of the database's schema
- A CORBA system with corresponding IDL files [OMG]
- A C function library
- A set of URLs and associated query parameters
- A set of XML DTDs and the path of the filesystem directory that will contain corresponding XML files

3.2.3 Semantics

The *semantics* of an interface is the specification of how the entities in the interface are supposed to correlate to:

- entities in some other system, or
- entities in the world we live in.

Example

Consider a table in our car dealership's customer database. The table is named *tblCustomers*, and has the following columns:

- cust_num: integer
- street: string(255)
- city: string(60)
- state: string(2)
- car_pref: integer FOREIGN KEY(tblCarTypes)

A statement of the table's semantics may look like the following:

"*tblCustomers* contains one record for each customer that our dealership has ever sold a car to. A record is only removed from this table if it is discovered to refer to the same person that another record in this table refers to."

"*cust_num* is a unique identifier for each customer record. No two customers share the same customer number. A customer number is not intended to correspond to any value outside of this database. For a given customer record, this value will never change."

"*street*, *city*, and *state* are the mailing address that the customer has most recently been known to live it. The *street* field includes an apartment number specification if needed."

"*car_pref* is a foreign key into the *tblCarTypes* table. This field shows what type of car the customer has most recently expressed preference for."

3.3 Semantic translation in data integration architectures

The problem of semantic differences arises when information from one information system is expressed in terms of the interface of another information system.

We'll use the term *semantic translation* to refer to the process of dealing with the semantic problems that occur when trying to express the information stored in one information system in the form used by another information system.

3.3.1 Asymmetry of semantic translation between information systems

Semantic translation between two information systems A and B can be usefully divided into two parts: The semantic translation required to allow information to flow from System A to System B, and the semantic translation required to allow information to flow from System B to System A.

Recognizing the potential asymmetry of information flow between two systems is useful for the following reasons:

- Saving humans' time A good semantic translation in one direction may not work in the other direction.

An example is when the source information system is more specific in certain details than the target information system. Moving data from the source to the target can be lossy without any damage to the quality of the integration.

If a programmer is assigned the task of writing a function to perform such a one-way translation, his task is fairly straightforward.

However, if a programmer had to write a translator that also moved information from the less-precise information system to the information system whose model semantics assumed all data was very precisely categorized, the programmer's job is much harder and perhaps more time consuming.

- Assisting in selection of an integration architecture Different integration architectures can result in different information flow patterns between the constituent information systems.

  A choice of integration architecture may be influenced by the architect's recognition that for two information systems being integrated, software supporting the information flow in one direction might be harder to develop than software that supports information flow in the opposite direction

3.3.2 Data warehousing: one-way information transfer

One-way transfer is typical of the ETL (Extract, Transform, and Load) stage of maintaining a data warehouse. See Section 2.4 for details.

Example

Consider an integration where information from an individual car dealer's inventory system is made available to the inventory system of the dealership's parent company via a daily upload of data. However, the parent company shares no inventory data with the individual dealership, and the parent company cannot query the dealership's inventory system.

In this case, some agent (either at the dealership or at the parent company) must translate the inventory information from the schema of the dealership into the schema used by the parent company.

For example, the dealership's inventory system may distinguish between "light pickup truck" and "heavy pickup truck", whereas the owning company only has one category for such vehicles: "pickup truck".

A one-way translation is all that's required here: The integration system must be able to represent information from the individual dealer's inventory system in terms appropriate for the owning company's information system, but not vice versa.

3.3.3 Virtual view / federated databases: Two-way information transfers

Two-way flows of application-specific information are typical of *virtual view* architectures (see Section 2.3) and of federated database systems (see Section 2.3).

Example

Consider the car dealership example above. Suppose that the parent company wanted the ability to query the car inventory of each car dealership it owns, so that the parent company can provide a very up-to-date inventory information on its company web site.

Somewhere in the system will probably be software that translates a query expressed in terms of the parent company's schema to a query expressed in terms of each individual dealership's inventory system.

For example, suppose a user of the parent company's web site requests a listing of all "pickup trucks" that are in stock at any of the owning company's constituent dealerships.

The parent company's web site will delegate this query to the inventory system of each the car dealerships it owns. However, those car dealerships' inventory systems don't know what a "pickup truck" is. Somewhere between the parent company's web site and the dealership's inventory system, software must run that reformulates the query to look for any "light pickup truck" or "heavy pickup truck" that's in stock.

This is an example of a two-way information transfer. The query, which contained application-specific information, flew from the parent company's information system to the dealerships' information systems. The results, which also contained application-specific information, flew from the dealerships' information systems to the parent company's information system.

3.4 Semantic problems in data integration

3.4.1 Simple type differences

System interfaces are usually composed of common-place elements, such as C it unsigned ints, SQL *date*s, and Java Strings.

Sometimes there can be a very simple correspondence between a data source's exposed interface and the integrated system's exposed interface.

For example, both systems might provide a mechanism that when given a car's VIN (Vehicle Identification Number), yields the date that the car was built.

Suppose that the data source's interface is a SQL database with a table that maps cars' VIN to the car's manufacture date (presented as a SQL *datetime*).

If the integrated system's exposed interface is written in Java, then it would be desirable for the integrated system to present a car's construction date as a *java.util.Date* object.

3.4.2 Unexpressed semantics of schemas

Properly understanding the semantics of the data sources interfaces is vital to integrating the data sources.

To illustrate this, consider what would happen if the developer who were integrating the data sources did *not* understand the semantics of the data sources' interfaces:

The developers, trying to make their integrated information system useful, would try to tell the users of the system how the data coming out of the integrated information system was supposed to correlate to the user's lives.

For example, "The field labelled 'Number sold: ' is the number of cars sold on the date that appears in the field labelled, 'Date', be all of our dealerships combined.

However, in order to be able to make such semantic claims about the outputs of the integrated system, the developers would need to know the semantics of the interfaces that the integrated system got that data from.

If the developers who are writing the software to integrate the data sources aren't able to understand the semantics of the data sources' interfaces, they can't justify any semantic claims about the integrated information system's interface semantics.

3.4.3 Lossy mappings

Sometimes information systems can use different levels of precision to describe the same entities. This causes problems when constructing an integrated system.

Example

Consider the customer databases of two car dealerships. In both databases is a record of the type of car that each customer prefers to drive. This is collected to help the dealership know how to advertise best to each customer.

At one dealership, the customer preference information is very detailed: A customer's preference is expressed in terms of the manufacturer and model line of the car the user likes best. For example, customer 'Charlie Brown' prefers 'Ford F150 pickup truck'.

At the other dealership, the customer preference information is less specific: All that can be specified is the general class of vehicle. For example, 'Lucy Brown' prefers 'pickup truck'.

It happens to be the case that every kind of vehicle described in the first dealer's customer preferences database can be cleanly mapped into a class of vehicle in the second dealer's database. For example, a 'Ford F150 pickup truck' is a 'pickup truck'.

However, the opposite is not true: A vehicle preference from the second dealership's database does not map cleanly into a vehicle preference from the first dealership's database. Therefore, the two databases' vehicle preferences have an *onto mapping.*

Now suppose that an organization tries to create in integrated system that draws customers' vehicle preference information from the two dealerships' customer databases. The developers of the integrated system are confronted with the *onto* relationship described above.

3.4.4 Different categorizations

Different information systems can record similar information in ways that are so different from each other that integrating the information can be very awkward.

Example

Suppose two car dealerships track the amount of gasoline used at the dealership each month.

One dealership records the monthly use in terms of volume (i.e., gallons) purchased per month.

The other dealership records the monthly use in terms of money spent on gasoline per month.

Now suppose that an integrated system is being developed to show the gasoline use from all car dealers in the larger organization. The developers of the integrated system must wrestle with the difference in measurements.

Note that it could be argued that these two values aren't legitimate candidates for integration, because they actually represent two different details about the dealerships. However, the reality is that the concepts are so similar that a developer might genuinely be asked to provide a (numerically) approximate integration of the values.

3.4.5 Recognizing object identity

Data sources can have an unstated assumption that there's a one-to-one correspondence between entities in the data source and entities in the outside world.

For example, a car dealership would ideally have only one "customer" record per actual human customer. This is an important quality of the system, because it allows users of the system to perform certain reasoning that otherwise wouldn't be sound.

However, what happens if two data sources being integrated might both have a record for the same customer?

If an integrated system makes a false assumption that the data sources have disjunct sets of customer records, then the integrated system now has duplicate customer records. Reasoning that assumes non-duplicate customer records would be impossible with the integrated system although it would be possible with any of the individual data sources.

3.4.6 Conflicting data

Interface semantics often make an implicit claim that data using the interface is absolutely correct. People usually know to take that claim with a grain of salt. When unifying data from two or more data sources, contradictions can be reached, leading to an internally inconsistent view of data.

Example

Suppose that two car dealership have, over time, both sold the same car to a customer.

Each dealership maintains an inventory database that records for each car ever held by the dealer, the following pair of values: Vehicle Identification Number, Date-of-manufacture.

At one dealership, the value was correctly entered:

"123456842", "February 14, 2000"

At the other dealership, the value was incorrectly entered:
"123456842", "April 1, 2000"
When these data are integrated into a single information system, the conflicting values are detected.

3.5 Approaches to resolving semantic issues in data integration

Numerous approaches have been proposed to facilitate the integration of existing and proposed data integration systems.

The automated creation of integrated information systems is an open problem, and may always require some human involvement to resolve integration issues.

However, ad-hoc solutions exist for many of the semantic problems mentioned earlier in this section, and theory is being developed that may eventually yield better tools for the task.

3.5.1 Intuitive ad-hoc solutions

While the researchers look for theoretically sound solutions, real organizations need to integrate systems as best they can. This can lead to dealing with problems in an ad-hoc manner.

Here are some simple approaches to the semantic problems described earlier in the section.

Simple type differences

These are perhaps the most benign problems to deal with during integration, because:

- the software needed to implement the conversion can probably appear in a very localized part of the integrated system's source code, and
- this is a kind of conversion that many other software developers are also likely to need to do. This implies that it's quite likely that conversion libraries will be available to the developers of the integrated system.

Unexpressed semantics of schemas

This is a problem that presently requires human involvement to sort out. Confirming the semantics of data sources may involve talking with developers who previously worked with the data sources, talking with users, and some guesswork.

Onto mappings

If uniformity of detail in the integrated system has a high priority, then probably the most reasonable solution is to have the integrated system provide only the subset of information about an entity that is available from all relevant data sources.

A more sophisticated integrated system might allow its interface to provide additional information about an entity in those special cases where the particular data source involved has more information than the common subset.

Different categorizations

This is a very messy problem. Acceptable solutions are likely to be very application-dependent.

Recognizing object identity

Some data sources might provide enough information to allow the integration software to unambiguously detect matches between entities.

For example, two data sources might both use a customer's Social Security Number as a customer key. This makes duplicate detection trivial.

Duplication detection may involve guesswork. Software systems are available that try to make good guesses about duplicate records based on the information available. A common application of this is removing duplicate entries when large mailing lists are merged.

Conflicting data

Various approaches might be reasonable depending on the situation:

- When a conflict is detected, bring it to the attention of a human. The human can look for problems such as data entry errors and make a judgement. This audit might also lead to a correction of the original data in one of the data sources.
- If one system is considered more trustworthy than the other, use the answer provided by the more trusted system.
- If more than two systems provided conflicting answers, treat each data source's answer as a vote.
- If the answer is a real number, then allow mathematical interpolation (perhaps the arithmetic mean) to be the final answer presented by the integrated system.

3.5.2 Generalized solutions

Generalized solutions to solving semantic problems in data integration appear to be entirely academic as of this writing, and appear to be limited in the semantic problems they can deal with.

Examples include SchemaLog [LSS93] and Enterprise Requirements Analysis [GL].

4 Querying the Integrated Data

The main purpose of building data integration systems is to facilitate the access to the multitude of data sources. The ability to correctly and efficiently process the queries to the integrated data lies in the heart of the system. The traditional way of query processing involves the following basic steps:

- getting a declarative query from the user and parsing it
- passing it through a query optimizer which produces an efficient query execution plan that describes how to exactly evaluate the query, i.e., apply which operators, in what order, using what algorithm
- executing the plan on the data physically stored on disk

The procedure described above also applies to query processing in data integration systems in general terms. However, the task is more challenging due to the complexities brought by the existence of multiple sources with differing characteristics. First of all, we need to decide which sources are relevant to the query and hence should participate in query evaluation. These chosen data sources will participate in the process by their own query processing mechanisms. Second, due to potential heterogeneity of the sources, there may exist various access methods and query interfaces to the sources. In addition to being heterogeneous, the sources are usually autonomous as well and therefore not all of the them may provide full query capability. Third, the sources might contain inter-related data. There may be both overlapping and inconsistent data. Overlapping data may lead to information redundancy and hence unnecessary computations during query evaluation. Especially in the case where there are large number of sources and the probability of overlap is high, we may need to choose the most beneficial sources for query evaluation. The last but not the least, the sources may be incomplete in terms of their content. Therefore, it may be impossible to present a complete answer to user's query. This list of complications is extensible.

As discussed in Section 2, a data integration system may be built in two major ways: by defining a mediated schema on the participating data sources without actually storing any data at the integration system (virtual view approach) or by materializing the data defined by the mediated schema at the integration system (materialized view approach). In both of the approaches, the user query is formulated in terms of the mediated schema. However, in the latter approach, since the data is stored at the integration system according to the mediated schema, query evaluation is no more difficult than traditional way of query processing. The major issue there, is the synchronization of data with the changes to the original data at the data sources, i.e., maintenance of the materialized views. We discuss this issue in Section 6. During maintenance, views defined on the data sources have to be processed on the data sources to re-materialize the updated data. In other words, query processing on the original data sources is realized usually "off-line". [1] On the other hand, in the

[1]For immediate view maintenance policy, it is actually "on-line".

virtual view approach, every time a user asks a query, source access is required. Therefore, query processing for the virtual approach includes the issues that would arise for the maintenance stages of the materialized view approach. In this regard, we discuss mainly the query processing problem for the virtual view approach in this section.

In this section, first we briefly discuss the modeling issues which forms the basis of all the following arguments. Then we present the main stages in query processing in data integration systems in order, namely, query reformulation, query optimization and query execution.

4.1 Data Modeling

Traditionally, to build a database system, we first model the requirements of the application and design a schema to support the application. In a data integration system, rather than starting from scratch, we have a set of pre-existing data sources which would form the basis of the application. However, each of these data sources may have different data models and schemas. In other words, each source presents a partial view of the application in its own way of modeling. In fact, if we were to design a database system for the application starting from scratch, we would have another model, which would have the complete and ideal view of the world. To simulate this ideal, we need to design a unifying schema in a single data model based on the schemas of the data sources being integrated. Then each source needs to be mapped to relevant parts of this unified schema. This single schema of the integrated system is called the "mediated schema". Having a mediated schema facilitates the formulation of queries to the integrated system. The users simply pose queries in terms of the mediated schema, rather than directly in terms of the source schemas. Although this is very practical and effective in terms of transparency of the system to the user, it brings the problem of mapping the query in mediated schema to one or more queries in the schemas of the data sources.

The below figure shows the main stages in query processing in data integration systems. There is a global data model that represents the data integration system and each of the data sources has its own local data model. There are two conceptual translation steps: (i) from the mediated schema to exported source schemas, (ii) from exported source schemas to source schemas. The difference comes from the data models used. In the former one, the user query is reformulated as queries towards individual sources, but they are still in the global data model. In the latter one, source queries are translated into a form that is understandable and processable by the data sources directly, i.e., data model translation is achieved in this latter step. These two steps are performed by the mediator and the wrapper components in the system, respectively. In this section, we will be focusing on the operation of the mediator and the details of the wrapper will be presented in Section 5.

As Figure 1.4 indicates, in addition to modeling the mediated schema, we need to model the sources so that we can establish an association between the relations in the mediated schema and the relations in the source schemas. This

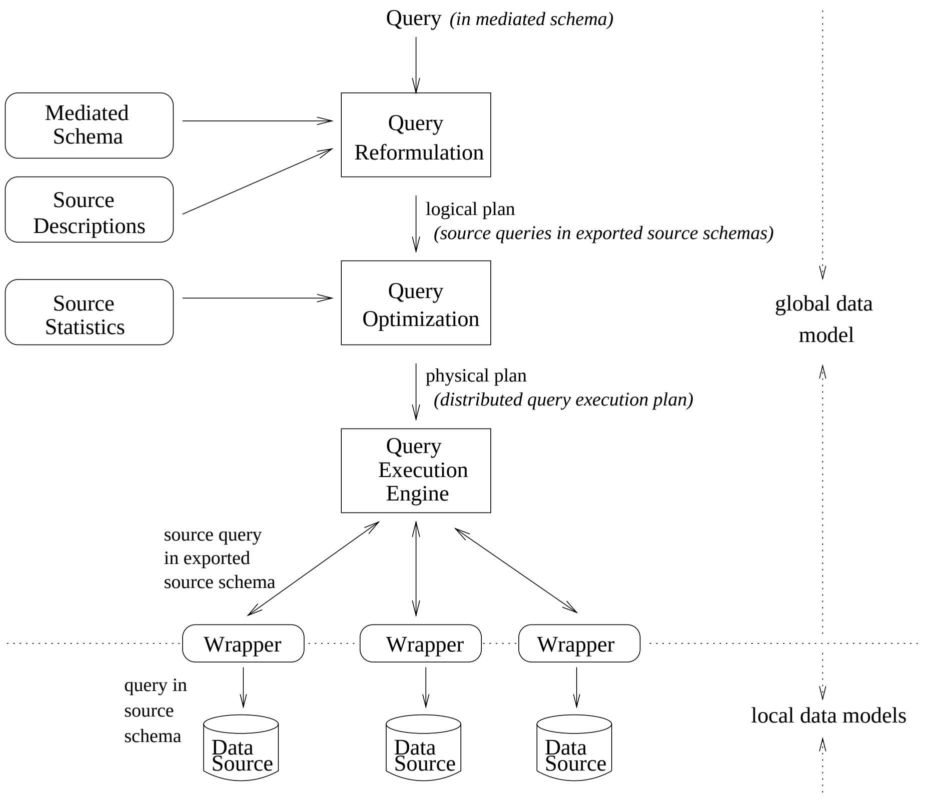

Figure 1.4: Stages of Query Processing

is achieved through source descriptions. The description of a source should specify its contents and constraints on its contents. Moreover, we need to know the query processing capabilities of the data sources. Because in general, information sources may permit only a subset of all relational queries over their relations. Source capability descriptions include which inputs can be given to the source, minimum and maximum number of inputs allowed, possible outputs of the source, selections the source can apply and acceptable variable bindings [].

To be able to present the methods for querying the integrated data, we need to choose a data model and language to express the mediated schema, source descriptions and the queries. Due to its simplicity for illustrating the concepts, we will be using relational model as our global data model and Datalog as our language.

4.1.1 Datalog

We can express queries and views as datalog programs. A datalog program consists of a set of rules each having the form:

$$q(\bar{X}) : -r_1(\bar{X_1}), \ldots, r_n(\bar{X_n})$$

where q and $r_1, \ldots, r_n$ are predicate names and $\bar{X}, \bar{X_1}, \ldots, \bar{X_n}$ are either variables or constants. The atom $q(\bar{X})$ is called the *head* of the rule and the atoms $r_1(\bar{X_1}), \ldots, r_n(\bar{X_n})$ are called the *subgoals* in the *body* of the rule. It is assumed that each variable appearing in the head also appears somewhere in the body. That way, the rules are guaranteed to be *safe*, meaning that when we use a rule, we are not left with undefined variables in the head. The variables in $\bar{X}$ are universally quantified and all other variables are existentially quantified. Queries may also contain subgoals whose predicates are arithmetic comparisons. A variable that appears in such a comparison predicate must also appear in an ordinary subgoal so that it has a binding.

.... explain the semantics of the rules, IDB, EDB predicates, conjunctive queries, recursive rules, etc ...

4.1.2 Modeling the Data Sources

To reformulate a query in mediated schema as queries on the source schemas, we need the relationship between the relations in the mediated schema and the source relations. This is achieved through modeling the sources using source descriptions.

There are three approaches to describing the sources:

Global As View (GAV) Approach

For each relation R in the mediated schema, a query over the source relations is written which specifies how to obtain R's tuples from the sources.

example will come here

This approach was taken in the TSIMMIS System [].

Local As View (LAV) Approach

For each data source S, a rule over the relations in the mediated schema is written that describes which tuples are found in S.

example will come here

This is an application of a much broader problem called "Answering Queries using Views". We will further discuss this problem in the next section.

One of the systems that used this approach was the Information Manifold System [].

Description Logics (DL) Approach

Description Logics are languages designed for building schemas based on hierarchies of collections. In this approach, a domain model of the application domain is created. This model describes the classes of information in the domain and the relationships among them. All available information sources are defined in terms of this model. This is done by relating the concepts defining the information sources to appropriate concepts defining the integrated system. Queries to the integrated system is also asked in terms of this domain model. In other words, the model provides a language or terminology for accessing the sources.

example will come here

This approach was taken in the SIMS System [].

Each of these approaches has certain advantages and disadvantages over the others. The main advantage of GAV is that query reformulation in GAV is very easy. Since the relations in the mediated schema are defined in terms of the source relations, it is enough to unfold the definitions of the mediated schema relations. Another advantage is the reusability of views as if they were sources themselves to construct hierarchies of mediators as in the TSIMMIS System []. However, it is difficult to add a new source to the system. It requires that we consider the relationship between the new source and all the other sources and the mediated schema and then change the GAV rules accordingly. Query reformulation in LAV is more complex. As we shall see in the next section, the most important work done on query reformulation focus on the LAV approach. However, LAV has important advantages compared to GAV: adding new sources and specifying constraints in LAV are easier. To add a new source, all we need to do is describe that source in terms of the mediated schema through one or more views. We do not need to consider the other sources. Moreover, if we want to specify constraints on the sources, we simply add predicates to the source view definitions.

Compared to GAV and LAV approaches, DL approach has the benefit of being more flexible.

... need to learn DL more to compare ...

4.1.3 Using Probabilistic Information

... will be written later ...

- for source completeness
- for overlap between parts of the mediated schema
- for overlap between information sources

4.2 Query Reformulation

Using the source descriptions, user query written in terms of the mediated schema is reformulated into a query that refers directly to the schemas of the sources (but still in the global data model). There are two important criteria to be met in query reformulation:

- Semantic correctness of the reformulation: The answers obtained from the sources will be correct answers to the original query.
- Minimizing the source access: Sources that can not contribute any answer or partial answer to the query should not be accessed. In addition to avoiding access to redundant sources, we should reformulate the queries as specific as possible to each of the accessed sources to avoid redundant query evaluation.

In this section, we will mainly discuss query reformulation techniques for the LAV approach of source modeling. The reason for this is that query reformulation in LAV is not straight forward and also it is one of the applications of an important problem called "Answering Queries using Views". In what follows, first we briefly summarize this problem together with its other important applications. Then we present various query reformulation algorithms for LAV.

4.2.1 Answering Queries Using Views

Informally, the problem is defined as follows: Given a query Q over a database schema, and a set of view definitions $V_1, \ldots, V_n$ over the same schema, rewrite the query using the views as Q' such that the subgoals in Q' refer only to view predicates. If we can find such a rewriting of Q into Q', then to answer Q, it is enough that we answer Q' using the answers of the views.

Interpreted in terms of the query reformulation problem for the LAV approach, this means the following: By using the views describing the sources in terms of the mediated schema, we can answer a user query written in terms of the same schema by rewriting the query as another query referring to the views rather than the mediated schema itself. Each view referred by the new query can be evaluated at the corresponding source this way. Basically we are decomposing the query into several subqueries each of which is referring to a single source.

Answering queries using views has many other important applications which include query optimization [], database design [], data warehouse design [] and semantic data caching []. For example, query optimization may be achieved by using previously materialized views for answering a query in order to save from recomputation. We are discussing data warehouse design issues in Section 6.

The ideal rewriting we expect to find would be an "equivalent" rewriting. However, this may not always be possible. In data integration systems in particular, source incompleteness and limited source capability would lead to rewritings that approximate the original query. Among the many possible approximate rewritings, we need to find the "best" one. The technical term for this best rewriting is "maximally-contained" rewriting. Note that we do not sacrifice from semantic correctness criterion here, rather we are preferring an incomplete answer to no answer at all. The below definitions formalize these terms:

Equivalent Rewritings Let Q be a query and $V = V_1, \ldots, V_m$ be a set of view definitions. The query Q' is an equivalent rewriting of Q using V if:

- Q' refers only to the views in V, and
- Q' is equivalent to Q.

Maximally-contained Rewritings Let Q be a query and $V = V_1, \ldots, V_m$ be a set of view definitions in a query language L. The query Q' is a maximally-contained rewriting of Q using V with respect to L if:

- Q' refers only to the views in V,
- Q' is contained in Q, and
- there is no rewriting Q_1 such that $Q' \subseteq Q_1 \subseteq Q$ and Q_1 is not equivalent to Q'.

A query Q' is contained in another query Q if, for all databases D, $Q'(D)$ is a subset of $Q(D)$. A query Q is equivalent another query Q' if Q' and Q are contained in one another.

4.2.2 Completeness and Complexity of Finding Query Rewritings

... will be written later ...

- source incompleteness
- recursive rewritings

4.2.3 Reformulation Algorithms

Given a query Q and a set of views $V_1 \ldots V_n$, to rewrite Q in terms of V_is, we have to perform an exhaustive search among all possible conjunctions of m or less view atoms where m is the number of subgoals in the query. The following algorithms propose alternative ways of finding query rewritings to avoid the exhaustive search.

The Bucket Algorithm (Information Manifold)
The main idea underlying the Bucket Algorithm is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each subgoal in the query in isolation, and determine which views may be relevant to each subgoal. Given a query Q, the Bucket Algorithm proceeds in two steps:

1. The algorithm creates a bucket for each subgoal in Q, containing the views (i.e., data sources) that are relevant to answering the particular subgoal. More formally a view V is put in the bucket of a subgoal g in the query if the definition of V contains a subgoal g_1 such that
   - g and g_1 can be unified, and

- after applying the unifier to the query and to the variables of the view that appear in the head, the predicates in Q and in V are mutually satisfiable.

  The actual bucket contains the head of the view V after applying the unifier to the head of the view.

2. The algorithm considers query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. For each possible choice of element from each bucket, the algorithm checks whether the resulting conjunction is contained in the query Q or whether it can be made to be contained if additional predicates are added to the rewriting. If so, the rewriting is added to the answer. Hence, the result of the Bucket Algorithm is a union of conjunctive rewritings.

example will come here

The Inverse-Rules Algorithm (InfoMaster)

The key idea underlying this algorithm is to construct a set of rules that invert the view definitions, i.e., rules that show how to compute tuples for the mediated schema relations from tuples of the views. One inverse rule is constructed for every subgoal in the body of the view. While inverting the view definitions, the existential variables that appear in the view definitions are mapped using Skolem functions to ensure that the value equivalences between the variables are not lost. The following examples illustrates the algorithm:

example will come here

In general, one function is created for each existential variable that appears in the view definitions. These function symbols are used in the heads of the inverse rules. The rewriting of a query Q using the set of views V is the datalog program that includes the inverse rules for V and the query Q.

The MiniCon Algorithm

MiniCon Algorithm looks at the problem from another perspective. Instead of building rewritings by combining rewritings for each query subgoal or mediated schema relation, we consider how each of the variables in the query can interact with the available views. This way the number of view combinations to be considered can be considerably reduced. The MiniCon Algorithm, like the Bucket Algorithm, first tries to identify which views contain subgoals that correspond to subgoals in the query. However, rather than building buckets, MiniCon Descriptions (MCDs) are built. MCDs are generalized buckets. Each correspond to a set of subgoals from the query mapped to subgoals from a set of views. First the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V. Then it looks at the variables that appear in join predicates in the query. The minimal additional set of subgoals that need to be mapped to subgoals in V given the partial mapping between g and

g_1 is found. These subgoals together with their mappings form an MCD. The following example clarifies the algorithm.

example will come here

The Shared-Variable-Bucket Algorithm

This algorithm, like the MiniCon Algorithm, also aims at recovering the weak aspects of the Bucket Algorithm to obtain a more efficient algorithm. Like the Bucket Algorithm, there are two steps: bucket construction and solution generation.

During the bucket construction step, Shared-Variable-Bucket Algorithm considers the equality constraints introduced by the "shared variables", i.e., variables that occur across multiple subgoals. Additional buckets are constructed called Shared Variable Buckets (SVBs) in order to handle the equality constraints. Each bucket contains only views that cover all the subgoals in which the shared variables representing the bucket appear.

In the solution generation step, a set of buckets is chosen such that each subgoal is represented by a single bucket in the set. From each bucket, a view is selected. Consequently, the solution to the query is expressed as a conjunctive query whose body is the conjunct of the selected views. The extra buckets ensure that the all generated solutions are sound solutions and this way the conjunctive query containment test at the end of the Bucket Algorithm is avoided.

example will come here

The CoreCover Algorithm

In this algorithm, closed-world assumption is taken where views are materialized from base relations. Among the possibly infinite number of rewritings, the aim is to find the ones that are guaranteed to produce an optimal physical plan if there exists any. Contrary to the other algorithms, this algorithm aims at finding equivalent rewritings rather than contained rewritings. Three different cost models are considered with the following motivations:

- Cost model M_1 tries to minimize the number of join operations
- Cost model M_2 additionally aims at minimizing the number of disk IO's by minimizing the size of the relations use in the plan
- Cost model M_3 aims at improving M_2 by dropping irrelevant attributes from the intermediate relations during evaluation.

We will be discussing the basic CoreCover Algorithm for the cost model M_1 and refer the interested readers to [] for modified versions developed for M_2 and M_3.

Intuitively, the first step in the algorithm is to find the set of query subgoals that can be covered by a view tuple, called "tuple-core". The second step is to find a minimum number of view tuples to cover query subgoals.

Rather than a technical discussion, we will present the algorithm with the following example:

example will come here

Comparison of the Algorithms

It is important that the algorithm scales well when the number of views increase.

after I write the examples ...

4.2.4 Alternative Query Reformulation for Dynamic Information Integration

... will be written later ...

4.3 Query Optimization and Execution

Query optimization refers to the process of translating a declarative query into an efficient query execution plan, i.e., a specific sequence of steps that the query execution engine should follow to evaluate the query. In addition to the operators and their application order specified in the query execution plan, the optimizer should also decide on the specific algorithms that implement the operators and which indices to use with them. There may be many possible execution plans. The best execution plan can be chosen in two ways: cost-based or heuristics-based. In the cost-based approach, the optimizer has to estimate the costs of candidate plans and choose the cheapest of them. Cost estimations are done using statistical information about the underlying data such as sizes of the relations and the selectivity of predicates. Heuristics-based plan generation involves using some rules of thumb like doing selections before joins. Usually heuristics-based technique is easier and cheaper than the cost-based one, because it does not need to consider and evaluate the cost of all possible plans. However, the optimal plan is not guaranteed.

As discussed in the previous section, query reformulation step already provides some optimizations on the query by pruning irrelevant sources and distinguishing the overlapping sources to avoid redundant computation. Furthermore, the rewritten queries are to be as specific as possible. However, these are logical or higher level optimizations. There are still many optimizations to be done when it comes to actually executing the logical plan generated by the reformulator physically on the data.

Query optimization in data integration systems is more difficult than the optimization problem in traditional databases because:

- Sources are autonomous. Optimizer may not have any statistics or either has few or unreliable statistics about the data stored in each of the sources.
- Sources are heterogeneous. They may have different query processing capabilities. The optimizer needs to exploit these capabilities as much as it can. In addition to what kind of queries the sources can process

and how they can process them, it is also relevant that what kind of processing power they have underlying their data management system and performance changes due to workload changes (??).

- In traditional databases, it is easy to estimate the data transfer time since it is between the local disk and the main memory. In data integration systems however, data transfer time is not predictable due to the existence of the network environment. Both delays and bursts may occur.

- On one hand, the sources are overlapping and there is redundancy for most of the time. That is why access to redundant sources should be minimized. On the other hand, some sources may become unavailable without any notice. Query optimizer should be able to handle these cases flexibly by replacing overlapping sources for each other to compensate for unavailability of any of them.

An additional problem that may cause inefficient query execution is that the logical plan produced by the reformulator tends to have a lot of disjunctions, i.e., union operations.

The bottom line is that it is difficult to decide statically what the optimum strategy would be to execute a query due to insufficient information and dynamicity of the environment. Therefore, the traditional approach of first generating a query execution plan and then executing it is no more applicable. [**?**] proposes an adaptive query execution approach in which query optimization and execution are interleaved. In this section we mainly discuss this approach.

4.3.1 Adaptive Query Execution

In addition to the above listed problems, [**?**] makes the following observations about query optimization in data integration systems:

- It is more important to aim at minimizing the time to get the first answers to the query rather than trying to minimize the total amount of work to be done to execute the whole query.

- Usually the amount of data coming from the data sources is smaller compared to case of querying a single source as in traditional database systems.

Adaptivity in [**?**] exists in two levels:

- interleaved planning and execution

- adaptive operators for execution engine

At a higher level, the former is achieved by creating partial plans called fragments rather than complete plans. The optimizer decides how to proceed next only after executing a fragment. Once a fragment is completed, the optimizer would know more about the sources and the environment so that it could do better planning for the rest of the query.

The latter includes using new operators during execution depending on the observations listed above. Two important operators used in [] are double-pipelined hash join and the collector operator.

Double-pipelined hash join is a join implementation that allows Tukwila to quickly return the first answers to the query in spite of the fact that some sources may be responding very slowly. In contrary to the conventional hash join where smaller of the two relations to be joined is chosen as the inner relation to hash by the join attribute, in double-pipelined hash join, both relations are hashed. This way, result tuples are produced as soon as the data from sources arrive. This masks the slow data transmission rates of some sources. The optimizer no longer has to make a decision about which relation should be the inner one (Normally, it would have to know the size of the relations to be able to choose the smaller one as the inner). Also, the processing is not blocked due to delays at the sources.

The collector operator is used to facilitate union over large number of overlapping sources. Using the estimates about the overlap relationships between the sources and depending on the run-time behavior of the sources (delays, errors) optimizer adapts its policy about how the unions should be performed and the collector operator achieves the application of this dynamic policy. Policies are specified using rules.

Both levels of adaptivity are realized through event-condition-action rules. Events are raised by execution of the operators or completion of some fragments and obtaining some partial results. When an event triggers a rule, first the associated condition is checked. If it is true, then the defined action is executed. Possible actions include reordering of operators, re-optimization, changing the policy of the collector operator and so on. The rules accompany the operator tree generated by the optimizer. They specify how to modify the implementation of some operators (for example, the collector) during run-time if needed and conditions to check at points where fragments complete in order to detect opportunities for re-optimization.

4.3.2 Query Translation

One thing we have treated as a black box until now is how actually the source queries in exported schemas (in schema of the sources but in the global data model) are translated into their actual schemas (in their local data models) and then get executed by their native query processors. This step is called the query translation step. It is achieved by the source-specific wrappers. Data extraction from sources by the wrappers is the topic of the next coming section.

5 Data Extraction

Data extraction deals with the issues arising during the process of getting data from the different sources to the integration system. It combines techniques from the areas of database systems and artificial intelligence (such as natural language processing and machine learning). In this section, at first we discuss *wrappers* that, as we have seen in the previous sections, are important components of the integration system. Then we review some work on the tools for semi-automatic and automatic wrapper generation, and conclude by discussing communication protocols used to access data sources on the network.

5.1 Techniques for Extracting Data (Wrappers)

During information integration from different heterogeneous data sources, we have to translate queries and data from one data model to another and from one data schema to another. As we mentioned in Section 2, this is done by wrappers that are written for each data source in the integration system. Each wrapper transforms queries in the unified schema to the queries in the format of the the underlying data source and then translates the results back to the unified schema. We would like to note, that mediator systems usually require more complex wrappers than do most warehouse systems.

5.1.1 Wrapper generation approaches

Wrapper designers can either construct the wrappers manually, or use the tools facilitating the wrapper code development. Three approaches are usually considered:

- **Manual**
  Hard-coded wrappers are often tedious to create and may be impractical for some sources. For example, in case of web sources: the number of them can be very big, new sources are added frequently and both the structure and the content of any source may change [AK97]. All these factors lead to the high maintenance costs of manually generated wrappers.

- **Semi-automatic (interactive)**
  It was noted that the part of a wrapper code which deals with the details specific to a particular source is often small [HGMN$^+$97]. The other part is either the same among wrappers or can be generated semi-automatically based on the declarative description given by a user. Techniques such as programming by example can be used for this purpose [].

- **Automatic**
  Tools for automatic wrapper generation can be site-specific or generic. They usually need training in the initial stage and are based on the supervised learning algorithms.

5.1.2 Tools for semi-automatic/automatic wrapper construction for structured/semistructured data

Here we review several techniques for semi-automatic wrapper generation.

- **Using formatting information in the semistructured pages on the web**

  The approach we describe here was designed for the case of web sources. As we mentioned above, writing wrapper code for web sources may be particularly hard due to the frequent changes of content and structure of the sources. On the other hand, data on the web often has a partial structure. That is, HTML documents often have some internal hierarchy of information, but this hierarchy is not specified explicitly. For example, a site of a travel agency may have information about several countries and hotels in the semistructured format. Some records, such as "a capital", "money units", "a language" will appear for all countries, while some others like "states" are country-specific. The presence of a partial structure in many web sources gives an integration system designer an opportunity to generate wrappers for the sources of a particular domain semi-automatically. Often this semistructured information may come to the web from the databases underlying the web sources. This raises the question of why could not we query these databases directly in this case? Unfortunately, a source may not set permissions for the outside users to query it for a number of subjective reasons.

  The approach was proposed by [AK97] and is used for semi-automatically generating wrappers for both *multiple-instance sources* and *single-instance sources*. Multiple-instance source contains information on several pages all of the same format. An example is cnn's weather pages for every city - pages for all cities have the same structure (for instance, there is always a **Current Conditions** section with temperature, humidity and wind specified). Wrapper must be able to answer queries about all sections of the individual page. Single-instance source contains a single page with semi-structured information.

  The authors identify three steps of a wrapper generation process for the types of sources mentioned above: "structuring the source; building a parser; adding communication capabilities between sources, a wrapper, and a mediator" [AK97] .

  – Structuring the source

    The first step refers to the finding heading tokens on a page, such as "Current Condition", "Temperature", "Wind", and organizing them in a hierarchy tree. Such sections are usually stressed in the document by the size of font (big), the type of font (bold, italic), by noticing a colon following such a token, etc. All these simple heuristics, used by the authors, proved to work well for the domains they specified. After a system has suggested the set of headings, a user may interfere by correcting the output. The hierarchy of the found headings is determined based on indentation spaces and font size. The grammar describing the structure of pages of a web source is produced as the result of this step. Results published by the authors show that usually just few corrections made by a user are needed for a web source.

  – Building a parser

    A parser for extracting any structured portion of data can be generated automatically given the output grammar of the first step.

  – Adding communication capabilities

    First, a wrapper needs some mechanisms to fetch the appropriate pages from the sources. In the case of a single page for each source, it is not a problem as long as URL of this page is known. In the case of multiple pages for a source, we need to map a query to the URL or set of URLs. In the case of the cnn weather site, for example, we can specify that for a given state and a city in it, the URL of the page containing the weather forecast can be obtained by adding the abbreviation of the region (for instance, *ne* stands for the north east), the abbreviation of the state and the name of the city to the end of the *http://www.cnn.com/WEATHER/* string. For example, the URL for Providence, RI is *http://www.cnn.com/WEATHER/ne/RI/ProvidencePWD.html.*

    Second, a wrapper relies on some protocols to deliver data over the network. We discuss some of these protocols in Section **??**. Authors of the paper were using Perl scripts.

    Third, a wrapper and a mediator need to communicate between themselves in the integrated system. In the reviewed system, a KQML language was used for this.

- **Template-based wrappers**

  The approach proposed by J. Hammer *et. al.* [HGMN$^+$97] is applicable to several types of data sources: relational databases, legacy systems, and web sources. Their wrapper implementation toolkit is based on the idea of *template-based translation.* A designer of a wrapper uses a declarative language to define the rules (templates) which specify the types of queries handled by a particular source. For each rule he also defines an action to be taken in case a query sent by the mediator of an integration system matches the rule. This action will cause a *native query* - a query in the format of the underlying source - to be executed.

  Filter queries are used to extend the set of queries a source can handle. If a source does not support some predicates, the query will be turned into two queries: the native query (that will contain only those predicates that are supported by the source) and the filtered query that will “postprocess” the results of the native query.

The process of query transformation consists of the following steps. First, a query from the mediator is parsed, then it is matched against the templates in the system. If the matching rule was found, the native query is processed by the data source, and the result is filtered with the filter query.

A rule-based language MSL [PGMA96] is used by the authors for query formulation. Below we give an example of an MSL query, a template matching it, and a corresponding native query and a filter query. For the purpose of the example (that is based on the example presented in [HGMN$^+$97]) the data source is a relation database.

Example. Let us refer again to the example of Acme Cars company that has a relation
Automobiles(vin, autoMake, autoModel, autoPrice, autoYear).
This relational database consisting of just one relation, is our data source. We need to write a wrapper supporting MSL queries to this source. We further assume, that the source does not support comparison predicates on the **autoPrice** attribute. Let a user A ask the query about all Honda cars for sale whose price is less than 12,000$:

C :– C:<Automobiles
{<autoMake “Honda”><autoPrice P>}>
AND LessThan(P, 12,000)

One of the templates, matching this query is:
C :– C:<Automobiles{<autoMake $A >}>

Notice, that the result of this template query is a superset of the results asked by the user query. The action corresponding to this template is to select all automobiles with the **autoMake = $A**. In the system, $A is substitued with “Honda” and a native SQL query for the relational database is produced:

SELECT *
FROM Automobiles
WHERE autoMake = “Honda”.

The only thing remained, is to postprocess the results of this query using the following filter query:

C :– C:<Automobiles{<autoPrice P>}>
AND LessThan(P, 12,000)

After that, the result can be returned back to the mediator of the integration system.

- **Inductive learning techniques for automatically learning a wrapper (using labeled data)**

  These techniques are sometimes called *wrapper induction* techniques [Eik99] and are based on inductive learning. According to [Eik99], *inductive learning* is the task of computing a generalization from a set of examples of some unknown concept. This generalization should suggest the model explaining all of the examples.

A very simple example of an inductive inference is when a teacher says a sequence of numbers: 2, 4, 6, 8, 10; and then asks a pupil to guess the rule he used to produce the next number from the previous (in this case $p_{n+1} = p_n + 2$.

Line Eikvil [Eik99] points out the following classification of the inductive learning methods used for wrapper induction:

– Zero-order learning
  They are also called *decision tree learners* as their solutions are represented as decision trees. The drawback of these methods is coming from the fact that they are based on the propositional logic that has a number of limitations. For example, they can not deal with several relations in a relational database [Eik99].

– First-order learning
  Methods of this type can deal with first-order logical predicates. *Inductive logic programming* is a method of this class, widely used due to the ability to deal with complex structures such as recursion. Two approaches - *bottom-up* and *top-down* - are often used as a part of the first-order learning.

  The bottom-up approach first suggests a generalization based on few examples. Then this generalized model is corrected based on the other examples.
  The top-down approach starts with a very general hypothesis and then distills it learning from negative examples.

Some known systems for inductive learning of wrappers are STALKER [MMK98] and the system described by N. Kushmerick *et. al.1¡* [KWD97]. An exellent overview of some other systems for information extraction by inductive learning is given in [Eik99].

5.2 Data Source Interfaces

The integration of information systems will almost always result in the development of software that accesses those information systems' public interfaces.

A plethora of options is available for how data sources expose their data to other computer systems. This subsection will explore some of the issues that differentiate various types of interfaces.

5.2.1 General issues with interfaces

- **Separation of interface into application-specific and reusable layers**

  Interfaces can often be divided into two parts:

  – a set of primitive components that is used in numerous applications, and
  – a set of application-specific components.

  This distinction is made in the OSI network protocol stack and elsewhere.

  For the remainder of this section, we'll use the term **primitive interface** to refer to these multi-application, reusable interfaces.

  For example, consider a car dealership's customer database system. Suppose that the system exposes its data via a CORBA interface.

  CORBA, and perhaps TCP/IP, could be considered the primitive components of the interface that appear in many applications. Thanks to the support of outside organizations, CORBA and TCP/IP are available for use in many different applications.

  However, the set of objects and methods exposed with CORBA for the customer database system is application specific. (For example, a Customer object or a Customer.scheduleForTuneup() method.) Few if any other applications are likely to ever use the same application-specific interface.

- **Resolution of data addressability**

  Primitive interfaces are often ignorant of the structure of the data that the interface helps to transfer between the interface user and the interface implementor.

  For example, one could move car inventory data over FTP in binary transfer mode. The design of FTP is such that the protocol doesn't interpret the structure of the transferred data when using binary transfer mode.

  However, it's possible that the structure of the file being transferred is very rich indeed. For example, the file could be a serialized Java object with a well-defined structure.

  This raises the issue of the resolution of addressability provided by an interface.

  In the example above, the FTP interface was capable of providing data addressability down to the file level.

  However, the FTP interface does not support the selection of a particular XML element from the XML document.

- **Data types**

  Primitive interfaces typically provide a set of one or more primitive data types that the interface explicitly recognizes.

  For example, SQL offers *varchar*, *int*, etc. C APIs offer *int*, *float*, *char**, etc.

  These primitive types may be given special treatment in the language bindings that let a programming language use the interface.

For example, big-endian and little-endian computers use different bit-level representations of integer data. The SQL bindings for programming environments on various computers will convert the bit-level representations of SQL *int* data into a format that's appropriate for the computer using the interface.

Application-level constructs must be expressed (directly or indirectly) in terms of these primitive data types.

- **Interface semantics**

  Primitive interfaces are designed with the intention that the primitive interface itself assigns no meaning to the particular data moving across the interface.

  A result of this is that primitive interfaces are powerless to express application-level semantics of the data.

  See Section 3 for more details on interface semantics.

5.2.2 Network vs. non-network primitive interfaces

In data integration, an important aspect of primitive interfaces is whether or not network communications are used.

- **Non-network interfaces**

  These interfaces do not explicitly support network communications. This is a problem because data integration may involve two or more computers that must communicate.

  If data integration requires that two or more computers communicate, but one or more data source does not offer a network-friendly interface, work-arounds may be painful:

  - Adding network connectivity may introduce yet more software to add network connectivity.
  - Using off-line data transfer mechanisms (tape, CD-RW, etc.) can lead to undesirable latencies in the transfer of data from a data source to the integrating system.

- **Network-capable interfaces**

  These interfaces explicitly support network communications.

  Unfortunately, many modern networks (both short-distance and long-distance) have a set of problems that can be difficult to deal with from a software level.

  - **Failures of network links**

    Various problems can and do occur in the connections between computers. In the modern Internet, it's common to hear of a backhoe operator accidentally digging through a bundle of optic cable used by an Internet service provider, or of a necessary router malfunctioning.

In naively integrated systems, a network link failure can cripple the entire integrated system or cause the loss of data.

– **Potentially independent failures of communicating applications**

  If the software for integrating data executes on a different computer than one of the data sources, it's possible for only one of the computers to fail.

  This possibility raises very similar problems in system design to those problems arising from fallible network links.

– **Potentially significant communications costs / performance issues**

  In long-distance communications between computer systems, transferring data is often expensive. Businesses that have multiple offices often lease expensive network connections to allow the computers at each site to communicate quickly with computers at other sites.

  Fiscal cost may need to be a consideration in system design when data integration involves computers at separate sites.

Examples

– **FTP** - *File Transfer Protocol*

  This protocol gives one computer access to part of another computer's file-system. It also offers a username/password form of access control.

– **CORBA** - *Common Object Request Broker Architecture*

  CORBA is a an architecture designed to allow computers to interoperate regardless of what operating systems, programming languages, network protocols, and application domains are involved [**?**].

  CORBA allows software applications to present their services to client applications. CORBA is an object-oriented system, so application services are represented as CORBA objects and the methods on those objects.

  IDL

  A CORBA server application must define the details of its exposed interface using *OMG IDL* (Object Management Group's Interface Definition Language).

  IDL defines objects and methods in a manner that's very syntactically similar to how classes or interfaces are defined in C++ and Java.

  IDL is programming language- and operating system-neutral, however. The basic data types available in CORBA IDL have well-defined bindings to various programming languages. This allows tools vendors, rather than application programmers, deal with the issues such portability of number and string representations.

IDL can be compiled into *client stubs* and *object skeletons*. Client stubs are language- / platform-specific bindings that application programs use to invoke the services of a particular CORBA object. Object skeletons are language- / platform-specific bindings that the implementer of a CORBA object must complete.

CORBA server applications (a.k.a. object implementations)

CORBA is like many other client/server protocols such as DCOM (Microsoft's Distributed Common Object Model), RPC (Remote Procedure Call), or Java RMI (Remote Method Invocation), in that it defines how an application can expose its functionality to other (possibly distant) applications.

To make a service available via application, a server application is developed that provides implementations for the object skeletons produced when the service's IDL was compiled.

At runtime, a CORBA server application registers its availability with a an Object Request Broker (see below) to make the server application to client applications.

CORBA client applications

Once a service's IDL has been compiled into appropriate client stubs, a client application can be written to invoke the stub's methods. The stub acts as a proxy. When the client application invokes a stub method (using the programming language's normal technique for calling methods / functions, the CORBA infrastructure will do the necessary work of translating that method / function call into the network messaging needed to invoke the corresponding method in the server application.

ORB (Object Request Broker)

We've seen how IDL is used to define the syntax of an interface that a server application wishes to expose. We've seen how, from a client's and server's perspective, the invocation of a stub method in the client somehow results in the invocation of a corresponding method in the server application.

An Object Request Broker is the piece of software that lets clients detect server's availability on the network, and that routes client request to the appropriate server.

– **ODBC** - *Open DataBase Connectivity*

  This protocol is designed to allow applications to access relational database data without regard to the vendor of the database.

  An application program using ODBC can formulate SQL statements to submit to a database server.

  ODBC provides the client application with the programming interface to submit the SQL statement to the server, and to retrieve the query results from the server.

In this sense, ODBC is like CORBA: It doesn't provide application semantics, but it does provide language bindings to the client application to simplify the invocation of (possibly remote) services.

6 Materialized View Management

A view defines a derived relation from a set of database relations. It is actually a query whose result is given a name that can be used like other ordinary relation names stored in the database. When the tuples of the virtual relation defined by a view are physically stored in a database, we call such a view *materialized view*.

Use of materialized views dates back to early 1980s [GM99]. They were first proposed to be used as a tool to speed up queries on views. Then they were also used to maintain integrity constraints and to detect rule violations in active databases. They gained serious reconsideration by the emergence of new applications like data warehousing. In this section, we discuss issues related to use of materialized views in data integration systems.

We presented the materialized view approach to data integration in Section 2. The term comes from the fact that a set of views are derived from the data sources and the answers to those views are actually stored in a repository called a *data warehouse*. The main purpose of this pre-computation is to improve query response time. None of the complex query processing steps described in Section 4 is needed to be able to answer a user query in data warehouses at the time the query is asked. Those steps are completed and the results are collected in advance of the queries. This not only provides the ability to answer many queries very quickly, but also increases the availability of the system since the warehouse continues to answer queries even if the underlying data sources may become inaccessible for some reason at the time of querying.

Of course the benefits mentioned above does not come for free. First, the views to be materialized need to be determined. Usually it is both costly and redundant to materialize all the derived relations defined by the views which constitute the unified schema of the integrated system. The most beneficial views need to be selected based on criteria like frequently asked queries. Second and more importantly, the views that are selected to be materialized need to be *maintained*. View maintenance refers to the process of synchronizing derived data stored at the warehouse with the updates on the base data stored at the underlying data sources. The naive way of maintaining views would be to re-materialize the views when a relevant update occurs. However, this is not desirable for good performance.

In this section, we first present the materialized view selection problem together with the proposed solutions. Then we discuss various approaches for maintaining the selected views efficiently.

6.1 Design and Selection of Views to Materialize

The problem of materialized view selection can be defined as follows: Given a set of queries to the integrated system with their access frequencies and a set of source relations with their update frequencies, find a set of views to be materialized such that the total query response time (i.e. query processing time) and the cost of maintaining the selected views are minimized [YKL97]. There may also be other resource constraints to be considered such as disk space, but the most important of all is the maintenance cost/time.

Previous research on this problem has concentrated on Multiple-Query Optimization (MQO) techniques. MQO is the problem of finding an optimal query execution plan for evaluating a set of queries simultaneously. Techniques used involve identifying the common subexpressions among queries, executing those once and reusing later. In general, there may be many possible plans for each query and there may also be many possible ways of combining them. Thus, the search space is really large. Two general approaches are: (i) producing local optimal plans for each query and then merging them, which does not guarantee an optimal solution, and (ii) generating a globally optimal plan, which has a larger search space. [SG90] has proven that MQO problem is NP-complete. Proposed solutions usually make use of heuristics to find a solution as close as possible to the optimal solution. The related work on materialized view selection follows a similar path.

[YKL97] proposes a method where a Multiple View Processing Plan (MVPP) is constructed from the set of queries. Then some parts of this plan are selected to be materialized. The cost comparisons are based on the following cost measures: Cost of a query is the number of rows in the table used to construct that query. Cost of query processing is the frequency of the query multiplied by cost of query access from materialized nodes. Cost of view maintenance is equal to the cost of constructing the view, i.e. re-materialization is assumed. Total cost is equal to the sum of the cost of query processing and the cost of view maintenance.

There are two stages to view selection:

1. finding a good MVPP
   MVPP is the global query execution plan in which local execution plans for individual queries are merged based on shared operations on common sets [YKL97]. There are two ways of finding the MVPP:

   - merging local optimal query plans
     Local optimal plans are computed for each query. Then the queries are ordered in a descending fashion based on their query processing costs multiplied by their access frequencies. If there are k queries, k MVPPs are constructed as follows:

     ```
     for i=1 to k do
         take the ith local query plan and
         incorporate all the others to it in order
     ```

     The view selection algorithm at stage 2 will be run on these k MVPPs and then the least costly one will be chosen. This approach takes linear time in terms of the number of queries.

   - generating a globally optimal plan
     Rather than the locally optimal plans, all possible plans for each query are considered. The problem is mapped to a 0-1 integer linear

programming problem which is stated as follows: Select a subset of the join plan trees such that all queries can be executed and the total query processing cost is minimum [YKL97]. Then the set of join trees found are used to construct the MVPP. Solution to the linear programming problem is the optimal solution. However, solving it is exponential in the number of queries. Therefore, usually near-optimal solution is found.

2. selecting views to materialize from the MVPP
   An execution tree is built for the given MVPP whose nodes correspond to intermediate results to the queries. We can simply choose the complete tree or all the leaf nodes for materialization. These correspond to materializing all the queries and all the base relations, respectively. However, our aim is to find a set of intermediate nodes to materialize such that the total cost for query processing and view maintenance is minimized. The brute force way of finding this set is to compare the cost of every possible combination of nodes. This is not efficient. We have to use some heuristics. The algorithm presented in [YKL97] is based on the following idea: Whenever a new node is considered to be materialized, we calculate the saving it brings in accessing all the queries involved, subtracting the cost for maintaining this node. If the value is positive, then this node will be materialized and added into the solution set.

A somewhat similar approach is presented in [Gup97] which is based on using greedy heuristics and AND-OR graphs. An AND-OR graph represents a set of query plans. AND-OR graphs of the queries are merged to obtain an AND-OR view graph. Each node in the AND-OR view graph represents a view that could be selected for materialization. The problem is to choose among the nodes of the AND-OR view graph such that sum of total query response time and total maintenance time is minimized. [Gup97] states that the minimum set cover problem can be reduced to this problem and it is NP-hard. A near-optimal algorithm is presented which uses greedy heuristics. The set of the selected views has a benefit and at each step views that would increase the benefit of this set would be added to the set. Special cases of AND view graphs, OR view graphs, view graphs with indices are also investigated in [Gup97].

[RSS96] and [MRRS00], which mainly focus on the view maintenance problem, indirectly cover some methods that are applicable to view selection. [RSS96] proposes to augment a given set of materialized views with an additional set of views that may reduce the total maintenance cost. The selection problem here is to determine the additional views. [MRRS00] applies MQO techniques both to view selection and maintenance. Selection comes into play where additional views are to be materialized temporarily for efficient maintenance. The claim is that the same techniques are also applicable to selection of permanent views to materialize.

There are also research studies in materialized view selection for the special case of data cubes [HRU96] and multidimensional datasets [SDN98] in OLAP. We do not present them in detail here.

6.2 The Problem of View Maintenance

Materialized views are derived from data originally stored at multiple data sources. As primary copies of data at the data sources get updated, materialized views become stale or inconsistent with the underlying data. We call the process of bringing the materialized views up-to-date with the changes in the underlying data *view maintenance*. A materialized view can always be brought up-to-date by re-evaluating the view definition. However, recomputing the views every time the base data changes is not very efficient. Besides, [GM95] points out that in general only a part of the view changes in response to changes in the base relations, which is called the *heuristic of inertia*. Thus, only the parts of the views that are affected from the changes need to be computed and updated. This is called *incremental view maintenance*. In this subsection, we present the dimensions of the problem and alternative policies for view maintenance. Incremental view maintenance techniques will be discussed in the following subsections.

6.2.1 Dimensions of the Problem

The following parameters determine the complexity of the view maintenance problem [GM99]:

- Available Information
  It refers to the amount of information available to the view maintenance algorithm. The view definition and the actual update occurred on base data have to be known to the algorithm. In addition to that, information like the content of the materialized views, the contents of the base relations, the definitions of other views and integrity constraints at the data sources might also be accessible to the algorithm. Depending on how much information is available, the task of view maintenance might be facilitated. For example, if we knew that a certain attribute is a key at the underlying data source, then we would know also know that every insertion would have a different value for that attribute. Hence, an insertion at the source would require an insertion at the materialized view that refers to that attribute.

- Allowable Modifications
  It determines what modifications can be handled by the view maintenance algorithm given other parameters. These might include insertions, deletions, updates, group updates, etc.

- Expressiveness of the View Definition Language
  View definition language may also facilitate or complicate the task of view maintenance. Views might be defined at various levels of expressiveness through languages including conjunctive queries, aggregation, recursion, negation, etc.

- Database and Modification Instance
  Current contents of the data sources or the materialized views and the modification may also determine the capabilities of the maintenance algorithm.

- Complexity
  A dimension that is somewhat at a different level than the others is the complexity dimension which refers to the efficiency of the view maintenance algorithm. Complexity can be measured in multiple sub-dimensions including complexity of view maintenance language, view maintenance algorithm or amount of extra information needed.

6.2.2 View Maintenance Policies

There are two main steps in materialized view maintenance: *propagate* and *refresh* [GM99]. Propagate step involves computing changes to be done on the materialized views upon changes to the base data and in refresh step the computed changes are actually applied on the materialized views. Propagate step always precedes the refresh step. The decision of *when* to perform the refresh step is called a *view maintenance policy.* Maintenance policies can be categorized as follows:

- Immediate View Maintenance
  Refreshing is done within the transaction that changes the base data. The advantages of this policy are that queries are processed fast and always return up-to-date results. The reason for this is that materialized views are brought up-to-date in advance of the queries. On the other hand, this policy slows down the transactions at the data sources since propagation and refreshing are to be done in the transaction's scope. Besides, this policy may not always be applicable when the data sources are fully autonomous and their commit decisions can not be delayed by the integrated system.

- Deferred View Maintenance
  Refreshing on the views is done later than the transaction that changes the base data. Log of changes to the base data are to be kept. This policy allows *batch updates* by applying all the changes collected in the log to the views at the same time. There are three deferred view maintenance policies:

  – Views are refreshed lazily at query time. It is guaranteed that query answers will be consistent with the base data and this is done without slowing down the transactions at the sources. However, queries to the integrated system are processed more slowly.

  – Changes to views are forced after a certain amount of change to the base data have been done. Both transaction and query performances are good, but queries may return non-up-to-date results.

  – Refreshing is done periodically in certain time intervals. This is also called the *snapshot maintenance*. Again, in spite of the good transaction and query time, queries may return non-up-to-date results.

In general, immediate maintenance does not scale with the number of materialized views, but deferred maintenance does. Therefore the decision of which views to maintain immediately has to be made very selectively. If real-time queries are asked on a view for which consistent results are crucial, then that view should be maintained immediately. Views which are queried relatively infrequently can be maintained in a deferred fashion. Usually decision support applications, where a stable copy of the derived data is more important than freshness, use periodical deferred policy. [CKL$^+$97] provides a decent study on consistency and performance issues in supporting multiple view maintenance policies. Materialized views that are related to each other may become inconsistent if they are maintained under different policies. Mutual consistency between views has to be settled.

The next question to ask is *how* view maintenance is applied. In the next subsections, we discuss how actually the maintenance should be performed.

6.3 Incremental View Maintenance

Incremental view maintenance algorithms have been investigated for a long time as an efficient alternative to re-materialization. Most of the work in this area consider the problem for centralized database systems where materialized views are used for purposes like speeding up queries on views or implementing rule checking efficiently. The problem has additional facets when considered in the scope of data integration systems. However, previous work still applies to some cases and form the basis of algorithms for data integration applications. We believe the following categorization of incremental view maintenance algorithms clarifies the link between the two cases:

- pre-update algorithms: maintenance is performed before the base relations have been actually updated, as in the case of immediate maintenance policy where maintenance is performed within the transaction that is updating the source.

- post-update algorithms: maintenance is performed after the transaction that updates the relevant base relations is over.

Previous methods that apply to centralized databases naturally involve pre-update algorithms because the base relations and the materialized views are parts of the same system. However, data integration systems have to use post-update algorithms since the underlying sources are autonomous and they are unaware of the maintenance procedures that are occurring in the integrated system. We can not force them to include maintenance procedures within their update transactions. As stated in [ZGMHW95], information sources are decoupled from the data warehouse. This brings additional problems about consistency.

In this section, our focus is on methods devised for incremental view maintenance in general. We present techniques specifically on data integration systems in the next subsection.

[GM95] provides a survey of incremental view maintenance algorithms classifying them according to view language and available information dimensions. Here we discuss some of them without giving an explicit classification. Our aim is to give a flavor of the important issues that are addressed in many of those algorithms.

[BLT86] handles Select (S), Project (P) and Join (J) views in isolation first and then considers them together as SPJ views. For each case, both insertions and deletions are considered. For S views, inserted tuples are simply unioned and deleted tuples are simply subtracted from the materialized view data set. Updating P views when deletion occurs in the base relation is more complicated. The problem stems from the fact that a tuple in the view that is projected on some particular attribute may be there due to multiple tuples in the base relations. If one of these base tuples is deleted, the derived tuple may not have to be deleted since there are other existing base tuples that it is derived from. This problem is solved by using *counter*s for view tuples. A view tuple would have to be deleted when the counter dropped to 0. Upon insertions of new tuples to one of the join relations, J views should only perform join between the newly added tuples and the other join relation rather than computing the join between two relations from scratch. Deletions are handled in a similar way by only joining the deleted tuples and then subtracting those from the original view. These methods are further combined together for SPJ views [BLT86].

[GMS93] presents two algorithms: *Counting* algorithm and *DRed* (Deletion and Re-derivation) algorithm. In both of these algorithms, the emphasis is on deletion since it is more problematic. Counting algorithm is proposed for non-recursive views with negation and aggregate functions. It is based on the *counter* method of [BLT86], but the view language is more general. For each tuple in the materialized view, number of alternative derivations is stored as the *count*. Relevant insertions increment the count and relevant deletions decrement the count by 1. When the count drops to 0, the tuple need not be stored in the materialized view any more. This algorithm also works with recursive views only if every tuple has finite number of derivations. DRed algorithm works for general recursive views with negation and aggregation. This algorithm involves three basic steps: (i) ignore the alternative derivations and put the view tuple into the delete set if it gets invalidated at least by one of its derivations, (ii) remove the tuples from the delete set if they have other derivations, (iii) compute the tuples to be inserted to the views due to insertions to base relations.

In addition to the algorithmic approaches as summarized above, there are algebraic approaches to incremental view maintenance. [GL95] presents an approach based on multi-set/bag semantics. All the arguments are based on the equivalence of bag-valued expressions. Bag algebra expressions are used to represent the materialized views. Given a transaction that changes the state of the database and a set of bag expressions, they try to derive delta expressions which represent how the bag algebra expressions need to be updated. The goal is to find a minimal set of such delta expressions. [GL95] also emphasizes that proper handling of duplicates is important for computing the aggregate functions (like averaging a list of values) correctly.

[RSS96] explores what additional views should be materialized for optimal incremental maintenance of a given materialized view. [MRRS00] generalizes this idea to how to maintain a set of views efficiently by using additional temporarily or persistently materialized views. Approach involves materializing common subexpressions between view maintenance expressions as in multi query optimization algorithms. We mentioned this approach before in this section as we presented the selection of views to materialize.

6.4 View Maintenance in Data Integration Systems

The main problem in data integration systems in terms of incremental view maintenance is that maintenance has to be done after the updates at the data sources have occurred. Later, when the maintenance has to take place, the integrated system may need to ask additional queries to the data sources when it does not have all the information needed to perform the maintenance. If the data sources continue to change in the time between the updates known by the integrated system and the maintenance time, then the additional queries will be answered according to the new state of the data sources which is different than the one at the time of initial update (i.e., the update which is trying to be fixed at the integrated system). This is called a *state bug* [CGL$^+$96] or *view maintenance anomaly* [ZGMHW95]. This problem stems from the fact that pre-update maintenance algorithms can not be used for data integration systems as they are. [CGL$^+$96] proposes two ways of avoiding the state bug:

- using the pre-update algorithms but restricting the updates and views so that correctness is guaranteed
- developing specific algorithms for the post-update case

[CGL$^+$96] proposes new algorithms for post-update case which are based on the usage of database invariants, i.e. conditions that are guaranteed to hold at every state of the database. These are used to maintain correctness. As [GL95], algebraic approach based on bag semantics is taken. [CGL$^+$96] also emphasizes the minimization of the view down-time. Usually views become inaccessible for queries during maintenance. [QW97] addresses this problem through a two-version no locking (2VNL) algorithm. Two concurrent versions of the materialized views provide continuous and consistent access to the warehouse during maintenance.

[ZGMHW95], on the other hand, is based on a pre-update view maintenance algorithm. The algorithm in [BLT86], which we briefly summarized in the preceding subsection, is used as basis. [ZGMHW95] proposes ECA (Eager Compensating Algorithm) in which extra compensating queries are used to eliminate anomalies. In fact anomalies would not occur if we re-computed the views or stored the copies of base relations referenced in the views, but both of these options are too costly and not good options compared to incremental view maintenance. In ECA, the basic idea is to send compensating queries to the data sources to avoid the potential anomalies that may occur according to query answers coming from the data sources. In other words, the warehouse eagerly forces data sources to send correct information. This is done by anticipating what kind of anomalies can occur beforehand and preparing view maintenance queries which contain compensating expressions in addition to the view maintenance expressions that would avoid the anomalies.

Next subsections discuss how the incremental maintenance process could be made more efficient.

6.5 Update Filtering

Not all the updates at the data sources cause updates at the materialized views. We can speed up the maintenance process if we can detect which base data updates have no effect on the views, and hence need not be maintained. Such updates are called *irrelevant updates* and the procedure of pruning irrelevant updates from the maintenance plan is called *update filtering*. [BCL89] calls queries/views that are not affected from the updates *queries independent of updates*.

Most of the work in this area aims at theoretically defining necessary and sufficient conditions for the detection of irrelevant updates for the cases of insertions, deletions and modifications [BLT86, BCL89, LS93]. [BCL89] defines irrelevant updates as update operations applied to a base relation has no effect on the state of a derived relation independently of the database state. [LS93] reduces the update independence problem to equivalence problem for Datalog programs and provides decidability results for different cases. [BLT86] presents more practical algorithms for detecting irrelevant updates. The views considered are in the form of PSJ queries. Selection condition is the primary determinant for deciding relevance. For insertions at the base relations, we substitute the values of the inserted tuple in the selection condition of the view. If the selection condition becomes unsatisfiable, then the insertion is irrelevant to the view, i.e., no tuple needs to be inserted to the view. Else, the insertion *may* be relevant to the view. Similarly, for deletions, we substitute the values of the deleted tuple in the selection condition of the view. If the selection condition becomes unsatisfiable, then the deletion is irrelevant to the view, i.e., no tuple from the view needs to be deleted. In general, satisfiability of boolean expressions is NP-complete. [BLT86] assumes boolean expressions that are conjunctions of inequalities. Then the problem can be solved in polynomial time. It can also be generalized to disjunctions of conjunctions, which adds a linear factor to the complexity.

In conventional database systems, update filtering can be implemented using integrity constraints or triggers. The base relations are not decoupled from the derived relations. View definitions are known to the whole system. However, in data integration systems, the filtering has to be done at the integration system level. Data sources can not perform filtering since they are not aware of the view definitions.

6.6 View Self-Maintenance

Another way to speed up maintenance is to minimize external data source access. As we mentioned earlier, to maintain a view, we may need to ask queries to the data sources in addition to the update information itself. This requires to communicate with the sources. We should try to exploit the information available at the integrated system (data warehouse) as much as we can to avoid this communication.

In general, *self-maintenance* refers to views being maintained without using all the base data. There exists different notions of its exact meaning depending on how much information is available. The ideal case is that the view update is performed locally at the integrated system by only knowing the particular base data update that has occurred, the view definitions and the materialized data. Whenever this is not possible, we need additional techniques to minimize base data access.

The first thing to do is to decide whether a given view is self-maintainable or not. If it is, then we need to know how to achieve self-maintenance. Otherwise, techniques may be developed to make it self-maintainable. Self-maintainability can be both investigated on a single view or on multiple views. Initially, we can consider each view in isolation. It should also be noted that self-maintainability is an issue specific to data integration systems. In traditional (centralized) databases, since all information is known to the system, there is no context for self-maintainability.

[GJM96] aims at defining self-maintenance rules for SPJ views. Self-maintainability algorithms are highly dependent on the view definition language. Three issues are investigated: (i) which relation is modified, (ii) what type of modification, and (iii) if key information can be exploited. The results they have come up with are as follows:

- For insertions, SP views are self-maintainable. SPJ views are self-maintainable only if join is a self-join (i.e. relation R is joined with itself) and join attribute is the key of R. Other SPJ views are not self-maintainable.

- For deletions, SPJ views are self-maintainable.

- For updates, if modeled as deletion followed by insertion, the rules for insertions and deletions hold. Otherwise, SPJ views are self-maintainable if updates are on non-exposed (i.e. not involved in any predicate in the view definition) attributes.

[BCL89] explores similar conditions for a more general view definition language.

[Huy97] investigates the meaning of self-maintainability at different contexts. They show that self-maintainability can be reduced to the problem of deciding query containment.

There are several techniques to make views locally maintainable [Huy97]:

- Multiple-View Self-Maintenance
  Views that are not self-maintainable when considered in isolation may become collectively maintainable at the integrated system when they are considered together [Huy97]. In other words, the information available to each view is extended to all the materialized views at the warehouse in addition to its own definition and materialization.

- Batch Updates
  Rather than maintaining each update operation separately, if we save the updates and maintain them all together, then the amount of work may be reduced. For example, if an update operation deletes a tuple and a following update inserts the same tuple back, then these two updates have no effect on the state of the materialized views when considered as a whole.

- Auxiliary Materialized Views
  By materializing additional views, other views may become self-maintainable. The basic idea here is to increase the amount of information available at the integrated system level.

Lastly, one important point to note is that self-maintenance also removes the situations where anomalies can occur when the maintenance is totally performed locally at the integrated system. The reason for this is that anomalies are caused by additional queries asked to the data sources some time after the related update has occurred. If a view is self-maintainable, then no additional querying is necessary.

6.7 Dynamic View Management

As stated earlier, materialized view management has two important components: view selection and view maintenance. Until now we assumed that views to materialize are selected once at the beginning according to some statistics on frequently asked queries and base data update frequencies and then the selection is over. From that point on, the system concentrates on the maintenance of those selected views. This kind of view management is called *static view management*. The major problem with this approach is that if the query workload or base data update patterns change, then the decisions about view selection become invalid.

The solution proposed in [KR99] is *dynamic view management* in which view selection and view maintenance stages are unified. The query workload is continuously monitored by the system and view selection decisions are updated dynamically. The constraints to be considered in addition to the changing workload patterns include disk space and maintenance window. Maintenance window has more importance than space because usually the system is unavailable for queries while the maintenance is being carried out. This time window has to be kept as short as possible. The more number of views materialized, the longer the maintenance window is. However, more materialization speeds up the query processing. Therefore, a compromise has to be made.

7 Systems

There exists many systems which intend to develop tools to facilitate the integration of both structured and unstructured data from heterogeneous data sources. In this section, we present some examples. We focus on some research projects, rather than commercial ones.

7.1 Mediated Systems

7.1.1 TSIMMIS

TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) consists of two main components: (i) the source specific translators (wrappers), and (ii) the *intelligent* mediators. Translators are responsible for converting a user query in the common global model into queries in the local models of the sources that the sources can execute and then converting the answer returned from the sources back to the common model. Mediators assemble information from sources, process and combine it, and transmit the final information to the end user.

In TSIMMIS, a simple-describing object model is used called the *Object Exchange Model (OEM).* OEM allows simple objects' nesting and all objects have *labels* to describe their meaning. OEM-QL query language is developed to request OEM objects. OEM-QL is an SQL-like language specified to deal with labels and object nesting. In TSIMMIS, both mediators and translators are automatically or semi-automatically generated from their high level request of the information process.

For interface, mediators and translators both take as input OEM-QL queries and return OEM objects. The good point here is that it allows new sources useful once a translator is supplied. There are two ways for end users to get information, one is to write applications that ask for OEM objects, the other is to use the browsing tool, named MOBIE (MOsaic Based Information Explorer), to specify queries using OEM-QL.

Another important issue in TSIMMIS is that there is no global schema. A mediator does not need to know details of all of the data it use. It is not necessary for any person or software component to have a global view of all the information managed by the system.

In TSIMMIS, constraint management is more difficult than the centralized systems. Usually they do not have transactions among different sources. Each source may have different policies to those data involved in a constraint. It is not guaranteed that consistent data will be accessed at each time it interacts with the system.

In summary, we can list three main differences between TSIMMIS and other systems:

- TSIMMIS concentrates on providing an integrated system which deals with very diverse and dynamic information.

- In TSIMMIS, information access and integration are intertwined.
- TSIMMIS requires more human participation.

For more information about the TSIMMIS project, please refer to [CGMH$^+$94].

7.1.2 SIMS

SIMS (Services and Information Management for decision Systems) is an information mediator for processing queries to multiple information sources. This system takes a domain-level query and dynamically chooses the useful sources, generates a query plan which describes the operations and some specific orders to deal with the data, and performs semantic query optimization.

The application domain models are defined by nodes, representing each class of objects, and their relations, defining relationships between the objects. Queries in SIMS are represented by the general domain model. The system translates the domain-level query into a set of source-level queries. The information source model define both the contents of the objects and their relationship.

To answer a query, SIMS first selects the appropriate information sources. The system provides a set of reformulation operators that are responsible for transforming the domain-level concepts into concepts that the information source could accept. The operators include Select-Information-Source, Generalize-Concept, Specialize-Concept, and Decompose-Relation.

The next step is to generate a query plan for the data process. The query plan defines the concrete operations that need to be executed and the order in which they will be executed. The system searches all possible plans with a best-first method until a complete one is found.

Finally, the system performs the semantic query optimization. "A set of applicable rules for the query is constructed. These rules would either be learned by the system or provided as semantic integrity constraints. Based on these rules, the system infers a set of additional constraints and merges them with the input query. The resulting query is semantically equivalent to the input query but is not necessary more efficient. The set of constraints in this resulting query is called the *inferred set*. The system will then select a subset of constraints in the *inferred set* to complete the optimization."

In summary, SIMS provides some ideas which are different from other integration systems:

- In SIMS, the integration problem is shifted from building a single integrated model to how to map between the domain and the information source models.
- The planning in SIMS is performed by an AI planner.
- Compared to other related works to search optimized queries, their algorithm considers "all possible optimizations by firing all applicable rules and collecting candidate constraints in an *inferred set*. Then the system

selects the most efficient set of the constraints from the inferred set to form the optimized subqueries".

Details about SIMS project can be found in [ACHK93].

7.1.3 ARIADNE

ARIADNE is an integration system, which is developed to extract, query and integrate data from semi-structured web sources.

ARIADNE project is based on an earlier work on the SIMS information mediator. However, web sources are different from databases in many ways, which means that it is required to construct new mediator query planning techniques to efficiently process web-based sources.

In ARIADNE, each web page was treated as a relational information source - a small database, thus it gives a simple and uniform representation to facilitate the integration of data.

In ARIADNE, query processing is decomposed into two phases: preprocessing phase and query planning phase. The ARIADNE source selection algorithm preprocess the domain model in order to efficiently select the sources, according to the classes and attributes in the query. ARIADNE uses a method named Planning-by-Rewriting to generate a plan. The initial, suboptimal plan is improved by applying rewriting rules.

Modeling the information on a page. In ARIADNE, most semistructured web pages are described as *embedded catalogs*, in which special markers are needed to locate information on a page. ARIADNE has a *demonstration-oriented user interface* for users to let the system know what information to extract from example pages. There is a machine learning system underneath the interface to induce grammar rules.

Modeling the information on a site - the connection between pages. To locate a page on a web site, the approach is to model the information needed to navigate through a web site. Then the planner can automatically decide how to locate a page. The developer uses the same approach as above to create wrapper for the index page, but this wrapper only wraps a single page. There are two common types of navigation methods used on web - direct indexing and form-based retrieval.

Modeling information across sites. Across different sites, the same entities may be referred to by different names. In ARIADNE, their approach is to choose a basic source for an entity's name and then apply a mapping from this source to each one of the other sources which have different scheme to name the entity. A mapping table is created for each entity in one data source, and if the mapping is computable, it will be represented by a mapping function, a program to convert one form into another form.

Details about ARIADNE project could be found in ARIADNE web page at [ARI].

7.2 Data Warehousing

7.2.1 WHIPS

There are two major components in data warehouse system: the *integration component*, and the *query and analysis component*. Compared to most commercial warehouse systems, which focus on the query and analysis component, WHIPS (WareHouse Information Prototype at Stanford) focuses on the integration component.

The system consists of dissimilar modules which could communicate with each other. Each module is implemented as a CORBA object, and each object has a set of methods available to other objects.

In WHIPS, the warehouse data is represented by the relational model, and views are defined in the relational model. The source monitor and wrapper are responsible for converting the underneath source data to the relational model before it is sent to any other module.

Each source is encapsulated by a source-specific monitor and wrapper. The functionality of monitor is to detect any modifications on the source data and inform the integrator of them.

Views are defined in a subset of SQL. They have Select-Project-Join views and aggregate views over all of the source data. Sometimes the view definition may also state in detail which algorithm to use for view consistency. Currently, they are adding simple SQL aggregate operators (min,max,count,sum, and average) to the view language.

The integrator is responsible for both the system startup, containing new source addition, and view initialization. But the main functionality of integrator is to assist view maintenance, by distributing different modifications used by different views. To do so, the integrator uses a set of rules which are generated automatically from the view tree when they are defined. The current integrator is carried out as an index over the view managers.

Each view has one view manager module, using one of the Strobe algorithms to keep the view consistency. There are two advantages to do that. First, the work can be done in parallel on different machines. Second, each view could make use of a distinct Strobe algorithm for its level's consistency.

"The query processor receives global queries from the view managers and poses the appropriate single-source queries to the source wrappers to answer them. it then passes the composite global query processor, performs distributed query processing, using standard techniques such as sideways information passing and filtering of selection condition to prune the queries it poses to the wrappers".

To sum up, WHIPS System allows views over heterogeneous sources and supplies incremental view maintenance in a modular and scalable fashion.

8 Open Questions and Research Issues

... will be formed later from everybody's list of research issues from the above

sections ...

9 Concluding Remarks

... will be written later ...

Bibliography

[ACHK93] Y. Arens, C. Y. Chee, C. Hsu, and C. A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Intelligent and Cooperative Information Systems (IJCIS)*, 2(2):127–158, 1993.

[AK97] Naveen Ashish and Craig Knoblock. Semi-automatic Wrapper Generation for Internet Information Sources. In *Second IFCIS International Conference on Cooperative Information Systems (CoopIS)*, Charleston, SC, 1997.

[ARI] http://www.isi.edu/ariadne.

[Ash00] Naveen Ashish. *Optimizing Information Mediators By Selectively Materializing Data*. PhD thesis, USC, March 2000.

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *Transactions on Database Systems (TODS)*, 14(3):369–400, September 1989.

[BDKV92] O. Buneman, S. Davidson, A. Kosky, and M. VanInwegen. A Basis for Interactive Schema Merging. In *Hawaii International Conference on System Sciences*, pages 311–322, 1992.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BLT86] J. A. Blakeley, P. Larson, and F. Wm. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., May 1986.

[BOT86] Yuri Breitbart, Peter L. Olson, and Glenn R. Thompson. Database Integration in a Distributed Heterogeneous Database System. In *ICDE*, pages 301–310, 1986.

[CGL$^+$96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *ACM SIGMOD International Conference on Management of Data*, pages 469–480, Montreal, Canada, June 1996.

[CGMH$^+$94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *10th Meeting of the Information Processing Society of Japan (IPSJ)*, pages 7–18, Tokyo, Japan, October 1994.

[CKL$^+$97] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting Multiple View Maintenance Policies. In *ACM SIGMOD International Conference on Management of Data*, pages 405–416, Tucson, AZ, June 1997.

[Eik99] Line Eikvil. Information Extraction from World Wide Web. A Survey, July 1999.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data Integration using Self-Maintainable Views. In *International Conference on Extending Database Technology (EDBT)*, pages 140–144, Avignon, France, March 1996.

[GL] Nectarios Georgalas and Pericles Loucopoulos. Integration of Business Operational Data using a Schema Integration Technique.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *ACM SIGMOD International Conference on Management of Data*, pages 328–339, San Jose, CA, June 1995.

[GM95] A. Gupta and I. S. Mumick. Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GM99] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, D.C., May 1993.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*, chapter 11: Information Integration. Prentice Hall, 2000.

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *International Conference on Database Theory (ICDT)*, pages 98–112, Delphi, Greece, January 1997.

[HGMN$^+$97] Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Marcus Breunig, and Vasilis Vassalos. Template-based wrappers in the TSIMMIS system. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[HM85] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD International Conference on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[Huy97] N. Huyn. Multiple-View Self-Maintenance in Data Warehousing Environments. In *International Conference on Very Large Data Bases (VLDB)*, pages 26–35, Athens, Greece, August 1997.

[JPSL$^+$88] G. Jacobsen, G. Piatetsky-Shapiro, C. Lafond, M. Rajinikanth, and J. Hernandez. CALIDA: A Knowledge–Based System for Integrating Multiple Heterogeneous Databases. In *Third International Conference on Data and Knowledge Bases*, pages 3–18, Jerusalem, Israel, 1988.

[KR99] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *ACM SIGMOD International Conference on Management of Data*, pages 371–382, Philadelphia, Pennsylvania, June 1999.

[KWD97] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *Intl. Joint conference on Aritificial Intelligence (IJCAI)*, pages 729–737, 1997.

[Lit85] W. Litwin. An Overview of the Multidatabase System MRSDM. In *ACM National Conference*, pages 495–504, October 1985.

[LS93] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *International Conference on Very Large Data Bases (VLDB)*, pages 171–181, Dublin, Ireland, August 1993.

[LSS93] Laks V.S. Lakshmanan, Fereidoon Sandri, and Iyer N. Subramanian. On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems. *DOOD'93*, December 1993.

[MMK98] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured web-based information sources. In *Conference on Automatic Learning and Discovery CONALD-98*, 1998.

[Mot99] Amihai Motro. Multiplex: A Formal Model for Multidatabases and Its Implementation. In *Next Generation Information Technologies and Systems*, page 138, 1999.

[MRRS00] H. Mistry, P. Roy, K. Ramamritham, and S. Sudarshan. Materialized View Selection and Maintenance using Multi-Query Optimization. Submitted for publication, March 2000.

[MW88] N. E. Malagardis and T. J. Williams, editors. *Standards in Information Technology and Industrial Control*, chapter Multidatabase Systems in ISO/OSI Environment, pages 83–97. North-Holland, Netherlands, 1988.

[OMG] www.omg.org.

[PGMA96] Y. Papakonstantinou, H. Garcia-Molina, and S. Abiteboul. Object fusion in mediator systems. In *International Conference on Very Large Databases*, Bombay, India, September 1996.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *ACM SIGMOD International Conference on Management of Data*, pages 393–404, Tucson, AZ, June 1997.

[Rea89] M. Rusinkiewicz and et. al. OMNIBASE: Design and Implementation of a Multidatabase System. In *1st Annual Symposium in Parallel and Distributed Processing*, Dallas, Texas, May 1989.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal, Canada, June 1996.

[SDN98] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized View Selection for Multidimensional Datasets. In *International Conference on Very Large Data Bases (VLDB)*, pages 488–499, New York City, NY, August 1998.

[SG90] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):262–266, June 1990.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases, September 1990. ACM Computing Surveys.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *International Conference on Very Large Data Bases (VLDB)*, pages 136–145, Athens, Greece, August 1997.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, CA, June 1995.