DYNAMIC DATABASE INTEGRATION IN A JDBC DRIVER

Terrence Mason
Iowa Database and Emerging Applications Laboratory, Computer Science
University of Iowa

Ramon Lawrence
Iowa Database and Emerging Applications Laboratory, Computer Science
University of Iowa

Keywords: integration, database, schema, metadata, annotation, evolution, dynamic, JDBC, conceptual, embedded.

Abstract: Current integration techniques are unsuitable for large-scale integrations involving numerous heterogeneous data sources. Existing methods either require the user to know the semantics of all data sources or they impose a static global view that is not tolerant of schema evolution. These assumptions are not valid in many environments. We present a different approach to integration based on annotation. The contribution is the elimination of the bottleneck of global view construction by moving the complicated task of identifying semantics to local annotators instead of global integrators. This allows the integration to be more automated, scaleable, and rapidly deployable. The algorithms are packaged in an embedded database engine contained in a JDBC driver capable of dynamically integrating data sources. Experimental results demonstrate that the Unity JDBC driver efficiently integrates data located in separate data sources with minimal overhead.

1 INTRODUCTION

Large-scale integration involving numerous heterogeneous databases is limited by imposing a static global view or requiring users to know the semantics of the integrated data sources. A robust integration system will dynamically update the global schema representing the integrated data sources. Our approach pushes the most challenging task of integration, identifying semantics, to local annotators instead of global integrators. The system automates the tasks of matching and resolving structural conflicts. Integrators are only responsible for annotating local data with the necessary context so that it may be integrated into the overall system. Annotation allows related concepts to be identified and referenced using accepted terminology.

The advantage of matching on schema annotations rather than the schemas themselves is that the overhead of human interaction is limited to gathering semantics at each database *individually* rather than identifying relationships during the matching process. This is important as the annotation only needs to be performed once per database, as opposed to having human involvement during every match operation. This reduces the requirement that global integrators fully understand the semantics of every schema.

The success of the integration rests on being able to construct standardized annotation that can be matched across systems. A common complaint against such an approach is that users will not adopt standards. Although this may be the case in some environments, it is not universally true. There is an increasing trend in organizations to standardize common terms, fields, and database elements, especially in the medical and scientific domains. For example, the National Cancer Institute Center for Bioinformatics (NCICB)[1] has developed a standardized ontology called Enterprise Vocabulary Services (EVS) (Covitz et al., 2003) for common terms in the cancer domain. NCICB has also created a Cancer Data Standards Repository (caDSR) to standardize the data elements used for cancer research. As part of a national initiative to build a grid of cancer centers, individual centers must conform their data to be compliant with Common Data Elements (CDEs) and EVS. This compliance is achieved by annotating their data sources with the accepted terminology. Thus, it is not uncommon to require a standard for participants who wish to share and integrate their data. Once local sources are annotated to conform to the standards, the global-level matching is considerably simplified because concepts will match across databases if they agree on the particular standardized term or data element.

[1]http://ncicb.nci.nih.gov

Mason T. and Lawrence R. (2005).
DYNAMIC DATABASE INTEGRATION IN A JDBC DRIVER.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 326-333
DOI: 10.5220/0002552103260333

Unity enables the dynamic integration of large numbers of data sources that are annotated according to a reference vocabulary. The architecture is capable of handling large-scale integrations in evolving environments, where the specific databases participating in the federation change frequently and their local schemas evolve over time. In addition, the driver's database engine integrates distributed data sources without requiring middleware or database server support and allows programmers simplified access to integration algorithms.

A lightweight integration system is implemented inside a JDBC driver to provide a standard interface for querying heterogeneous databases. Integration is supported with an embedded database engine that joins data located in multiple data sources into a single result. The JDBC standard allows the execution of queries in a general programming environment by providing library routines which interface with the database. In particular, JDBC has a rich collection of routines which make the interface simple and intuitive. The result is increased portability and a cleaner client-server relationship. For an integration system in an evolving environment to succeed, it will require an automatic and scalable approach with a standard interface to access the data.

The challenge of integration is to build a platform-independent system that supports automated construction of an integrated view, handles schema evolution, and allows the dynamic addition and deletion of sources. Our contribution is an integration architecture that has several unique features:

- A method for annotating schemas using reference ontologies and automatically matching annotated schemas to produce a global view.
- A simple conceptual query language that reduces the complexity often found in SQL queries.
- The system automatically determines the necessary joins and relations in each data source and across data sources.
- A system for tracking data provenance and detecting inconsistent data across databases.
- Updates the global view automatically to handle addition or deletion of data sources.
- Algorithms implemented in the standard JDBC interface.

The organization of this paper is as follows. Section 2 reviews existing integration systems. In Section 3, the JDBC driver implementation and its architecture is described. Section 4 details performance experiments that show the JDBC implementation has minimal overhead for integrating data sources. The paper closes with future work and conclusions.

2 PREVIOUS WORK

Several data integration prototypes (Goh et al., 1999; Kirk et al., 1995; Li et al., 1998) have been developed. In the literature, there are two basic types of data integration systems (Halevy, 2001; Ullman, 1997): global as view (GAV) systems and local as view (LAV) systems. The difference between the two types is how the mappings are expressed between the global view and the source. Although schema matching systems (Rahm and Bernstein, 2001) semi-automate mapping discovery between the global view and the sources, they have proven less successful in constructing the global view itself. While progress continues to be made using a reference ontology (Dragut and Lawrence, 2004), global view construction and its evolution remains an open issue.

Data integration systems (Lenzerini, 2002) require a global view to be constructed before local sources are integrated into the federation. The bottleneck in the integration process is the construction of the global schema, as schemas do not contain enough information to allow conflicts between data representations to be automatically identified and resolved. Current approaches to integration (Lenzerini, 2002) assume static schemas, data sources, and global level knowledge to build the integrated view. In practice, integrators have limited knowledge of the data sources. Systems must be integrated quickly, and the scale of integration makes defining global relations challenging.

Many commercial systems are based on federations (Sheth and Larson, 1990) such as federation in DB2 (Haas et al., 2002). These systems allow the sharing of schemas across databases but require users to perform the integration through query formulation. Federation is not desirable for large-scale integrations because of the complexity in writing queries. Such queries are susceptible to evolution and require the user to understand the semantics of all schemas.

Using an ontology for integration has been used previously (Collet et al., 1991; Decker et al., 1998). These systems require sources to completely commit to a global ontology and manually map all of their data to the ontology. Since ontologies have more powerful modeling constructs, their construction is more challenging, and it becomes an issue if the local sources will commit to using such an ontology. It remains challenging to construct and refine a global ontology. Unity does not require complete conformance to an ontology, but only a sharing of standard terms for common elements. ***No previous integration system has been deployed in a standard API such as JDBC.***

3 INTEGRATION ARCHITECTURE

The unique features and algorithms of Unity are implemented in a JDBC driver allowing programs to transparently query one or more databases. The ability to integrate databases without additional software makes it easier and faster to integrate systems in dynamic environments. The driver isolates the complexities of integration from the application programmer by automatically building an integrated view and mapping conceptual queries to SQL. While the system is deployed in the Unity JDBC driver (Figure 1), the algorithms are not specific to a JDBC implementation and can be deployed using other technologies. The TPC-H[2] schema is partitioned into two schemas (Figure 2) to provide an example integration scenario for this paper. A program invoking standard JDBC method calls will be used to describe the Unity JDBC Driver system architecture and integration algorithms. The functions are exactly the same as those in a single source JDBC driver. The difference is that the Unity driver allows the programmer to transparently access multiple sources instead of just one.

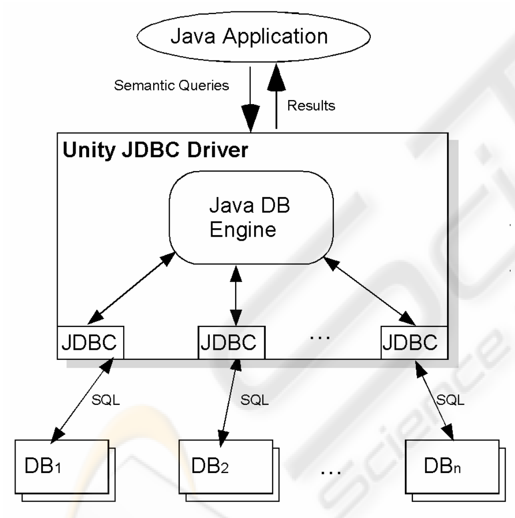

Figure 1: JDBC Integration Driver Architecture

3.1 Building the Global Schema

Given the importance of names and text descriptions in schema matching, it is valuable for schemas to be *annotated* before they are integrated. Our system uses annotation to build the global view in a bottom-up fashion. A data source administrator annotates a schema with meaningful names (possibly

[2]http://www.tpc.org/tpch

Part Database
part(p_partkey, p_name, p_mfgr)
supplier(s_suppkey, s_name, s_nationkey)
partsupp(ps_partkey,ps_suppkey)
nation(n_nationkey, n_name, n_regionkey)
region(r_regionkey, r_name)

Order Database
customer(c_custkey, c_name, c_nationkey)
orders(o_orderkey, o_custkey, o_orderdate)
lineitem(l_orderkey,l_partkey,l_suppkey,l_linenum,l_qty)
nation(n_nationkey, n_name, n_regionkey)
region(r_regionkey, r_name)

Figure 2: Database Schemas

using an ontology such as EVS) and exports the annotated schema in an XML document. The JDBC driver loads the individual annotations, matches names in the annotations to produce an integrated view, and then identifies global keys for use in joins across databases.

In the cancer domain (discussed in Section 1), matching on annotations is trivial as each common data element has a unique id and name. In other domains, the matching may be more complex, especially if standard conformance is loosely enforced. It is important to emphasize that the annotation, not the local or global schemas, is built according to an accepted standard. Unlike ontology-based systems, sources do *not* commit to a standard ontology/schema, but agree to annotate their existing schemas in a standardized fashion. The global schema is built by matching annotations in a bottom-up fashion and will contain only the concepts present in the sources.

Figure 3 shows a partial annotation of the *Part* relation in the *Part* database of Figure 2. The field *p_partkey* is annotated as *Part.Id* in the global schema. Two fields match if they have the same semantic name. This provides the mechanism for matching fields with the same domains across multiple databases. The primary and foreign keys are used to identify potential joins. The annotation provides the global attribute names for the query language. This flexible, scaleable, and unobtrusive approach succeeds as a site must *annotate* only local schemas so that they can be integrated with other systems.

The global schema produced by matching the *Part* database with the *Order* database is shown in Figure 4. The global schema is a universal relation of all attributes in the underlying schemas and is produced by taking the attributes of each schema and merging them into the universal view. Since attribute domains are uniquely identified by the annotation, this merge process is completely automatic.

Mismatches are possible at the global level if the annotators assign an incorrect term to their attribute. *Note that we expect there to be some errors during the*

```
<TABLE>
    <semanticTableName>Part</semanticTableName>
    <tableName>part</tableName>
    <FIELD>
        <semanticFieldName>Part.Name</semanticFieldName>
        <fieldName>p_name</fieldName>
        <dataTypeName>varchar</dataTypeName>
        <fieldSize>55</fieldSize>
    </FIELD>
    <FIELD>
        <semanticFieldName>Part.Id</semanticFieldName>
        <fieldName>p_partkey</fieldName>
        <dataTypeName>int</dataTypeName>
        <fieldSize>10</fieldSize>
    </FIELD>
    <PRIMARYKEY>
        <keyScope>5</keyScope>
        <keyScopeName>Organization</keyScopeName>
        <FIELDS>
        <fieldName>p_partkey</fieldName>
        </FIELDS>
    </PRIMARYKEY>
</TABLE>
```

Figure 3: Partial Annotation of Part Table

automatic matching because annotations may not be perfect. The global view is iteratively refined to correct such errors. However, the system is resistant to such errors as either a match goes undiscovered (creates two attributes in the global view instead of one) or a single attribute in the global view is used for two or more distinct attribute domains. When these errors are detected at the global level, they are resolved by modifying the local annotations. The result of merging annotations is independent of the order that they are merged. Note that the M-N relation *PartSupp* is abstracted from the global view as it will only be used during the local inference process.

```
Part.Id, Part.Name, Part.Manufacturer
Supplier.Id, Supplier.Name, Supplier.Nation.Id
Order.Id, Order.Customer.Id, Order.Date
LineItem.Linenumber, LineItem.Order.Id
LineItem.Quantity, LineItem.Part.Id, LineItem.Supplier.Id
Customer.Id, Customer.Name, Customer.Nation.Id
Nation.Id, Nation.Name, Nation.Region.Id
Region.Id, Region.Name
```

Figure 4: Global Schema

Note that the Unity JDBC driver could be used even when no annotation is performed and conceptual querying is not used. In this case, all relations from all databases are imported into the global view but not matched. Integration occurs when users explicitly state the joins in their queries. Thus, at the lowest level, the JDBC driver functions as a standard federated system allowing distributed access to the data sources. However, its true benefit is abstracting away the challenges of building joins and matching schema constructs manually.

3.2 Embedded Integration Engine

A lightweight database engine is developed and embedded in the JDBC driver implementation. It is required to parse the conceptual query, build a logical query plan, and then execute the query plan. The optimizer selects the join method and a join ordering to connect data across the data sources. The execution engine is capable of handling data sets larger than main memory by filtering and joining them efficiently using secondary storage at the client.

3.2.1 Making Connections

In the sample application (Figure 5), the URL specified in Line 1 references the `sources.xml` file containing information on each data source to be integrated. In Line 6, after registering the driver, the first task is to specify *which sources* to use. This is done using a sources XML file that contains source information such as the server, connection protocol, and annotation. This information is used by the Unity JDBC driver to connect to individual sources. The programmer may dynamically change which sources are included in the integration. When a connection is made, two things happen. First, the Unity JDBC driver uses the configuration information to make a connection to each data source specified. Second, the integration algorithm is run to match and merge the schema annotations in the XML files. The output after making a connection is an integrated, global view.

3.2.2 Conceptual Querying

Although querying is possible through SQL in this model, it requires complete knowledge of the relationships within and between the data sources to specify joins. A simpler conceptual query language is developed that allows users to query without specifying joins, as most users will not know all the details of an integrated schema. The query language is an attribute only version of SQL, where the SELECT clause contains the concepts to be projected in the final results and the optional WHERE clause specifies selection criteria for the query. By giving each field a meaningful semantic name, users can query on the name in the annotation instead of the explicit relation and field names of the individual data sources. Notice the absence of the FROM clause, as well as, no explicit specification of joins. The integration system will automatically identify these elements from the attributes specified in the conceptual query.

The concepts in the query (semantic names) map to fields in the relations. In Lines 8, 9 and 10 of the example program, the conceptual query (containing only attributes) is passed to the driver using the standard JDBC method. In this example, the user has queried on: Part.Name, LineItem.Quantity, and Customer.Name. These are mapped to their locations in the databases: Part.Name → part.p_name (Part DB), LineItem.Quantity → lineitem.l_qty (Order DB), Customer.Name → customer.c_name (Order DB). Users query on the annotation (attributes of the universal view), and the system performs query inference to find the required joins.

The conceptual query is parsed inside the Unity driver and converted to a parse tree using a JavaCC[3] based parser. The parse tree identifies all of the attributes and their relations in the local data sources for use in the generation of the integrated result.

```
import java.sql.*;
public class JDBCApplication
{
    public static void main(String[] args)
    {
        String url = "jdbc:unity://sources.xml";                          (1)
        Connection con;                                                    (2)
        // Load UnityDriver class
        try { Class.forName("unity.jdbc.UnityDriver"); }                   (3)
        catch (java.lang.ClassNotFoundException e) { System.exit(1); }     (4)
        try { // Initiate connection                                       (5)
            con = DriverManager.getConnection(url);                        (6)
            Statement stmt = con.createStatement();                        (7)
            ResultSet rst = stmt.executeQuery("SELECT Part.Name,           (8)
                    LineItem.Quantity, Customer.Name                       (9)
                    WHERE Customer.Name='Customer_25'");                   (10)
            System.out.println("Part , Quantity, Customer");               (11)
            while (rst.next())                                             (12)
                System.out.println(rst.getString("Part.Name")              (13)
                    +","+rst.getString("LineItem.Quantity")                (14)
                    +","+rst.getString("Customer.Name"));                  (15)
            con.close();                                                   (16)
        }                                                                  (17)
        catch (SQLException ex) { System.exit(1); }                        (18)
    }
}
```

Figure 5: Sample JDBC Application

3.2.3 Building Subqueries

The query's parse tree is translated into an execution tree consisting of operators such as select, project, join, and sort. The joins required to relate the attributes and complete the query must be identified. There are two levels of query inference required for querying a global schema: local and global. Local query inference determines the joins within a single data source. The conceptual query specifying attributes of the global schema is mapped to attributes and relations in each data source. A join graph G=(V, E) provides a directed graph representation of a relational schema, where each node in V represents a relation and each edge of E represents a foreign key constraint between two relations. Inferring the required joins reduces to the problem of determining the minimal edge set to connect the specified nodes (relations) in the graph. The minimal set of joins represents the strongest relationship between the attributes (Goldman et al., 1998). The inference algorithm uses a shortest-path approximation of the Steiner tree algorithm (Wald and Sorenson, 1984). The result of local inference is a join tree for each data source.

Since there is only one attribute/relation required from the Part database, it is straightforward to generate the join tree of one node. However, in the Order database, fields from the relations *customer* to *lineitem* are requested, but these relations do not have a direct connection between them. Query inference determines the shortest path which results in a join path from *lineitem* to *customer* through the *orders* relation. The join trees are used to build the execution trees by adding in any additional join operators required for the query. The selection operator must be added to the execution tree of the subquery for the Order database to select only `Customer_25` from the *customer* relation. The operator execution tree for the discussed subqueries are shown in Figure 6. The subquery execution trees are converted into SQL subqueries to be executed at each local data source by inserting the nodes in the FROM clause and the joins in the WHERE clause. In addition, the selection criteria is included in the WHERE clause. Note that both subqueries include the additional attribute `partkey`, which will be used to integrate the subqueries. The resulting subqueries are shown below:

Subquery 1: Order Database

```
SELECT l_qty, c_name, l_partkey
FROM lineitem, customer, order
WHERE o_custkey=c_custkey and l_orderkey=o_orderkey
c_name='Customer_25'
```

Subquery 2: Part Database

```
SELECT p_name, p_partkey FROM part
```

3.2.4 Integrating Results

Once local inference is complete and the subqueries are created, the database engine determines how the data extracted from each source can be related. To combine the subquery results, global inference determines the joins across data sources by identifying and

[3]https://javacc.dev.java.net

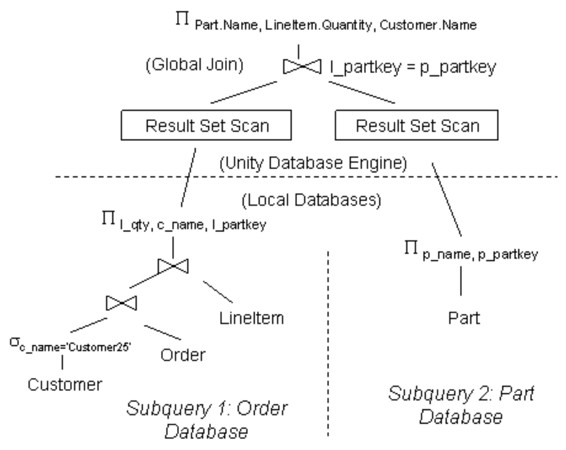

Figure 6: Operator Execution Tree

relating global keys. A *global key* is a key that identifies an object beyond the scope of the database. For instance, a book ISBN is a key that can be used to match book records across databases. In the annotation of Figure 3, global keys are identified by specifying the *scope* where they are valid. The primary key for the *Part* relation has the scope of the organization that owns the database. A *global join* is found between two databases if the key fields match and the scope of the global keys is compatible. Since both of the databases contain the *Part.Id* key with a shared domain and a valid scope, *Part.Id* is the global key used to join the subquery results. If no global joins exist, the user could explicitly specify global join conditions or a cross-product is performed. The plan of execution for the global query is represented in the operator tree shown in Figure 6.

The driver implements the execution tree by executing the two subqueries through JDBC drivers to the data sources to retrieve both ResultSets. The ResultSets are combined based on the *Part.Id* global join. Notice that *l_partkey* and *p_partkey* are included in the subqueries in order to execute the global join *p_partkey* = *l_partkey*. A global ResultSet is then constructed, which is accessed by regular JDBC methods, as shown in lines 12-15 of the example program.

The embedded database engine uses the ResultSetScan operator to scan in a JDBC ResultSet for each subquery one row at a time. The subqueries are threaded and executed in parallel. The optimizer pushes as much work as possible to the individual database sources, so that only the necessary global joins and filtering/sorting is performed at the client. Joins in the embedded database engine are performed using either sort-merge join or a variant of adaptive hash join. The database engine will use secondary storage to handle large joins that do not fit in memory efficiently. The global join of the sample query is processed by the database engine using a variant of adaptive hash join (Zeller and Gray, 1990).

3.3 Supporting Evolution

The driver supports two types of evolution: addition/deletion of data sources and schema evolution within a data source. Evolution is supported because all mappings are calculated dynamically: mappings from global concept names to fields in local sources, joins between tables in each local database, and global joins connecting databases. Annotating schemas before they are integrated reduces naming conflicts and allows the renaming of schema constructs. Since joins are calculated within a data source at query-time, structural evolutions such as promoting a 1:N relationship to a M:N relationship are handled. Finally, the conceptual query language on the global schema only references attributes, so if the global schema evolves, the query system will update the mappings without affecting existing queries.

3.4 Detecting Inconsistent Data

The architecture is designed to detect inconsistent data across data sources. It is common that names, addresses, and other text fields may have distinct values in several databases. For every data item produced in the global result, the *provenance* of the data item can be maintained. This allows a global user to determine the data source for each data element and the properties of that data source. It also allows the system to detect when two data sources have different values for the same field and present those differences to the global user. The user can then choose which data value to accept. Support for inconsistent data is unique to our architecture as other data integration systems assume their is no inconsistency between sources. Due to the cost of tracking data provenance, the user has the option of disabling this feature.

4 EXPERIMENTAL EVALUATION

The speed of global view construction and support for data source evolution is tested in the first experiment. In this experiment multiple TPC-H schemas are integrated. The results in Figure 7 show the times to integrate schemas (create the global schema), connect to all databases, parse the conceptual query, and generate the subqueries. The process for integrating schemas into a global schema is a linear time process based on the number of schemas integrated. The test is representative as integrated schemas will likely have data elements that share a large portion of the same domain.

Schemas	Integrate Schemas	Connect	Parse Subqueries	Total Time
2	0.219	0.203	0.110	0.532
3	0.235	0.234	0.125	0.594
10	0.485	0.437	0.406	1.328
100	1.375	1.719	5.063	8.157

Figure 7: Integration of TPC-H Schemas in Seconds

The time to parse the global query into subqueries has a greater complexity and processing time due to global inference. However, the time to parse and infer 100 SQL subqueries is completed in five seconds. The test results show that the Unity driver is capable of handling rapid evolution of the global schema, either within a single data source or by the addition or removal of data sources, as schema rebuilding is extremely fast. One hundred medium sized schemas may be integrated in less than 1.5 seconds and connected in three seconds. Note that the global view is created only once, whereas the parsing of the conceptual query and subquery generation is done for each global query.

In order to evaluate the query execution times for the Unity driver duplicate copies of the partitioned TPC-H benchmark databases with a 1 GB database size were installed on two identical computers running Microsoft SQL Server. A third identical computer executed the Java test programs, where each time was averaged over five query executions. The query execution times shown in Figures 8 and 9 are described below:

- **TPC-H** - Conceptual query executed through Unity driver against a single source TPC-H database.
- **JDBC TPC-H** - SQL query equivalent to conceptual query executed directly through SQL Server JDBC driver on a single source TPC-H database.
- **Partitioned Separate Computers** - Conceptual query executed on TPC-H data set partitioned into the Part and Order databases shown in Figure 2.
- **Partitioned on One Computer** - Conceptual query executed on TPC-H data set virtually partitioned into the Part and Order databases shown in Figure 2 on a single computer.

Two similar queries with contrasting result set sizes are used to evaluate the time required for Unity to process a conceptual query including execution time. The *large* (Figure 8) conceptual query SELECT p_name, l_quantity, c_name with equivalent SQL query[4] returns 6,001,215 tuples (equivalent to

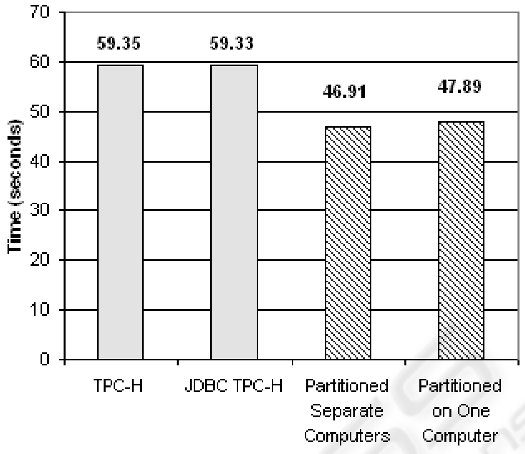

Figure 8: Large Part-LineItem-Customer Query

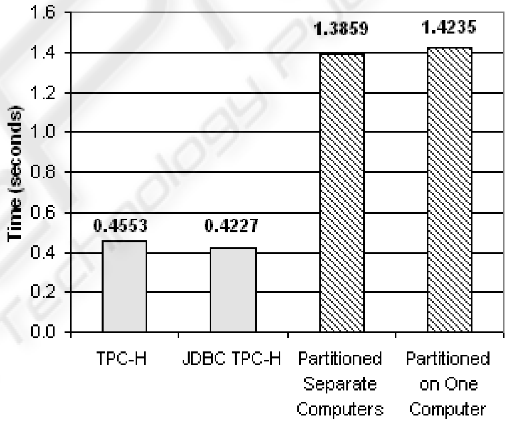

Figure 9: Small Part-LineItem-Customer Query

Lineitem table size). The *smaller* (Figure 9) query shown in the example program of Figure 5 generates the same results limited to Customer_25 for a result size of only 76 tuples[5]. The overhead for Unity to convert the conceptual query (TPC-H) into the equivalent SQL query listed (JDBC TPC-H) requires less than 0.025 seconds (25 ms) for either query. This demonstrates the nearly negligible time to pass a conceptual query through the Unity driver on a single source.

[4]*SELECT p_name, l_quantity, c_name FROM part, lineitem, customer, orders WHERE p_partkey=l_partkey and o_orderkey=l_orderkey and c_custkey= c_custkey*

[5]*SELECT p_name, l_quantity, c_name FROM part, lineitem, customer, orders WHERE p_partkey=l_partkey and o_orderkey=l_orderkey and c_custkey= c_custkey and c_name = 'Customer#000000025'*

The Figure 8 query execution times for the partitioned databases executed faster than the same query executed on a single source TPC-H database. The difference is due to the fact that the embedded database in Unity executes one of the joins on the client machine, as opposed to the server performing all three joins. The query executing all three joins on the server involves the join of the large *Lineitem* relation requiring full use of the CPU on the server. For the partitioned databases Unity is able to start the global join once the smaller ResultsSet from the *Part* relation is received. This allows for the embedded database in Unity to work in parallel with the server yielding an improved execution time for this particular query.

The results in Figure 9 show a longer execution time for integration versus full execution on the single source TPC-H database. The increased time is due to the time to transport to the client the entire *Part* relation (1.3 seconds) from the Part database in order to complete the global join. The single source query imports only the 76 tuples in the final result. Also, the results for the integration of the partitioned data located on one computer takes slightly longer than data partitioned on different computers, as the same resources are shared on a single computer.

5 CONCLUSION

The Unity JDBC driver provides an automatic and scalable approach to integrate and then query multiple data sources. The JDBC interface provides standard methods to access the data. The ability to quickly recompute the global view allows for dynamic integration of a large number of databases. The lightweight database engine embedded in the driver integrates the data from multiple databases transparently. The addition of the conceptual query language allows queries to be specified on the global view without the requirement of understanding the structure of each underlying schema. Experimental results show that this approach causes minimal overhead in the querying process. In addition, the driver efficiently executes the queries by identifying the subqueries to execute on the servers and using the client to complete the integration. This unique approach allows integration to be more automated, scaleable, and rapidly deployable.

Future work will investigate a more powerful global query optimizer. By obtaining information about the data sources including selectivity and relation size, the global join strategy could be optimized. In addition, strategies to effectively implement GROUP BY queries will be examined.

REFERENCES

Collet, C., Huhns, M., and Shen, W.-M. (1991). Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, 24(12):55–62.

Covitz, P., Hartel, F., Schaefer, C., Coronado, S., Fragoso, G., Sahni, H., Gustafson, S., and Buetow, K. (2003). caCORE: A common infrastructure for cancer informatics. *Bioinformatics*, 19(18):2404–2412.

Decker, S., Erdmann, M., and Studer, R. (1998). ONTOBROKER: Ontology based access to distributed and semi-structured information. In *Database Semantics - Semantic Issues in Multimedia Systems*, volume 138 of *IFIP Conference Proceedings*. Kluwer.

Dragut, E. and Lawrence, R. (2004). Composing mappings between schemas using a reference ontology. In *ODBASE*.

Goh, C., Bresson, S., Madnich, S., and Siegel, M. (1999). Context Interchange: New Features and Formalisms for the Intelligent Integration of Information. *ACM Transactions on Information Systems*, 17(3):270–293.

Goldman, R., Shivakumar, N., Venkatasubramanian, S., and Garcia-Molina, H. (1998). Proximity Search in Databases. In *VLDB*, pages 26–37.

Haas, L., Lin, E., and Roth, M. (2002). Database Integration through Database Federation. *IBM Systems Journal*, 41(4):578–596.

Halevy, A. (2001). Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294.

Kirk, T., Levy, A., Sagiv, Y., and Srivastava, D. (1995). The Information Manifold. In *AAAI Spring Symposium on Information Gathering*.

Lenzerini, M. (2002). Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246.

Li, C., Yerneni, R., Vassalos, V., Garcia-Molina, H., Papakonstantinou, Y., Ullman, J., and Valiveti, M. (1998). Capability Based Mediation in TSIMMIS. In *ACM SIGMOD*, pages 564–566.

Rahm, E. and Bernstein, P. (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350.

Sheth, A. and Larson, J. (1990). Federated Database Systems for Managing Distributed, Heterogenous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236.

Ullman, J. (1997). Information Integration Using Logical Views. In *ICDT'97*, volume 1186 of *LNCS*, pages 19–40.

Wald, J. and Sorenson, P. (1984). Resolving the Query Inference Problem Using Steiner Trees. *TODS*, 9(3):348–368.

Zeller, H. and Gray, J. (1990). An adaptive hash join algorithm for multiuser environments. In *VLDB 1990*, pages 186–197.