# Integration and Virtualization of Relational SQL and NoSQL Systems including MySQL and MongoDB

Ramon Lawrence
Department of Computer Science
University of British Columbia
Kelowna, BC, Canada, V1V 2Z3
ramon.lawrence@ubc.ca

***Abstract*—NoSQL databases are growing in popularity for Big Data applications in web analytics and supporting large web sites due to their high availability and scalability. Since each NoSQL system has its own API and does not typically support standards such as SQL and JDBC, integrating these systems with other enterprise and reporting software requires extra effort. In this work, we present a generic standards-based architecture that allows NoSQL systems, with specific focus on MongoDB, to be queried using SQL and seamlessly interact with any software supporting JDBC. A virtualization system is built on top of the NoSQL sources that translates SQL queries into the source-specific APIs. The virtualization architecture allows users to query and join data from both NoSQL and relational SQL systems in a single SQL query. Experimental results demonstrate that the virtualization layer adds minimal overhead in translating SQL to NoSQL APIs, and the virtualization system can efficiently perform joins across sources.**

**Keywords:** NoSQL, Big Data, MongoDB, integration, virtualization

## I. Introduction

There is a revolution going on in database systems. For years, database systems have been primarily based on the relational model, queried using standardized SQL, and accessed using common interfaces such as ODBC and JDBC. Although each relational system vendor has minor variations in implemented features, SQL dialect, and system interfaces, relational database systems are relatively interchangeable due to their common acceptance of standards. Recently, numerous NoSQL ("Not Only SQL") systems have been released and widely adopted in many domains. NoSQL systems have been developed to support applications not well served by relational systems, often involving Big Data processing. NoSQL systems can be categorized as key-value stores, document stores, and graph databases. Importantly, there are no common standard APIs for accessing the different NoSQL systems or standard query languages such as SQL. Typically, users must interact with these systems at a programming level with custom APIs. This reduces portability and requires system-specific code.

Although most NoSQL systems do not support SQL, there is no fundamental reason why they could not. The "NoSQL" label is a misnomer. The value of these systems has nothing to do with SQL support, but rather on their different architectural design decisions in order to achieve scalability and performance. SQL is valuable as it is a standard that allows portability, expressiveness, and has a massive installed base of trained users. As NoSQL systems evolve, there has been recognition of the value of SQL, and several commercial systems are hybrids combining a SQL relational processor with a Big Data (MapReduce) query processor. However, many of the most popular NoSQL databases, such as MongoDB, do not have SQL interfaces for their systems.

There has been prior research work on standardizing APIs and using SQL with NoSQL systems. A common programming API allowing querying Redis, MongoDB, and HBase was described in [1]. A SQL engine was integrated on top of HBase in [2], which used the Apache Derby query engine to process joins and other operators not supported by HBase. In [3], a SQL interface over SimpleSQL was defined. Partiqle [4] provides a generic SQL interface over key-value stores.

The contribution of this work is a generalizable SQL query interface for both relational and NoSQL systems called Unity. Unity allows SQL queries to be automatically translated and executed using the underlying API of the data sources (NoSQL or relational). Unity is an integration and virtualization system as it allows SQL queries that span multiple sources and uses its internal query engine to perform joins across sources. The key features of Unity are:

- A SQL query processor and optimizer including support for push down filters and cross-source hash joins.
- A customizable SQL dialect translator that uses mappings to compensate for different SQL variants in relational systems.
- A SQL to NoSQL translator that performs a mapping from relational operators in a query plan to NoSQL programming APIs. Relational operators not supported by the NoSQL system are performed using the Unity virtualization engine.
- SQL function translation allows functions to be mapped between different system implementations (e.g. different order of parameters, different names).
- Data virtualization allowing queries and joins across both relational and NoSQL sources.

978-1-4799-3010-4/14 $31.00 © 2014 IEEE
DOI 10.1109/CSCI.2014.56

The organization of this paper is as follows. In Section 2 is background on relational and NoSQL systems. Motivational use cases are provided in Section 3. The Unity architecture including details on the query processor/optimizer and implementation of the SQL translator is in Section 4. Section 5 presents experimental results, and the paper closes with future work and conclusions.

## II. Background

Relational systems have been the dominant form of data storage and manipulation for the last 30 years. The relational model [5] is both theoretically grounded and efficient to implement. In combination with standards such as SQL for querying and interface standards such as ODBC [6] and JDBC [7], relational systems are applicable to a wide variety of data problems. Relational vendors primarily differentiate products based on cost versus performance and system administration features. Applications written for one database system are generally portable to other systems, although translating stored procedures and system-specific features is more challenging.

The dominance of relational systems has been challenged in the past by object-oriented databases [8] and XML [9]. Object-oriented databases brought advantages like inheritance and type hierarchies and promised a better solution to the impedance mismatch problem where programmers must convert data in relational tables and attributes into program variables and objects. However, relational vendors modified relational systems with object-oriented features and extended SQL to capture most of the advantages of object-oriented systems while preserving the high performance and strengths of the relational model and SQL. Similarly, XML has a different model for data representation and exchange. Relational vendors extended systems to allow columns storing XML data and querying using XML languages like XQuery [10].

NoSQL systems have been proposed to tackle applications and problem domains poorly served by relational databases. These domains are primarily Big Data domains involving web data such as supporting millions of interactive users or performing analytics on terabytes of data such as web logs and click streams. The data in these domains is semi-structured, variable, and massive. Creating a suitable relational model may be difficult. There are several different types of NoSQL systems [11] including key-value stores (e.g. HBase[1]), document stores (e.g. MongoDB[2], CouchDB[3]), MapReduce systems (e.g. Pig[4], Hive[5]) and graph databases (e.g. Neo4J[6]). Key-value stores provide the simplest interface allowing storing and retrieving values using a hash interface. Document stores allow a structured document to be attached to a key. For MongoDB, the representation format is BSON (Binary encoded JSON).

[1]http://hbase.apache.org
[2]http://www.mongodb.org
[3]http://couchdb.apache.org
[4]http://pig.apache.org
[5]http://hive.apache.org
[6]http://www.neo4j.org

MapReduce systems, typically based on Hadoop, allow for large-scale processing of massive data sets on a cluster. There have been a variety of systems built on top of Hadoop including Pig and Hive to make it easier to construct queries rather than writing code. For instance, Hive supports HQL, a variant of SQL. Developers have also extended MapReduce systems by supporting SQL including HAWQ[7], HadoopDB/Hadapt[8], and Phoenix[9], a SQL interface for HBase. These systems often combine relational parallel processing (MPP) technology with Hadoop to speed up query processing.

NoSQL systems are often open source and are designed to handle larger data volumes at better performance than relational systems, although there is a debate on their relative performance [12]. Performance comparisons of NoSQL and SQL systems were done in [13]. The other major advantage, especially for systems like MongoDB, is better support for programmers. MongoDB allows a JSON object to be stored which can be easily converted into JavaScript objects in code. This simplicity and flexibility makes using MongoDB easier for many programmers without the need to understand SQL.

There has been some prior research in standardization for a variety of NoSQL systems including the SOS Platform [1] that defined a common API for Redis, MongoDB, and HBase. A standard programming API makes it easier to switch between NoSQL systems and improves programmer productivity, but does not address compatibility with existing query and reporting software that assume SQL-based access. In [3], a relational layer supporting SQL queries and joins is added on top of Amazon SimpleDB. SQL query conversion was done using regular expressions and rules rather than using a query parser and optimizer. In [14] a framework for representing NoSQL data in a relational model is presented as well as a system to map SQL queries to NoSQL systems. The prototype system used PostgreSQL and MongoDB and allowed joining relational data with data in MongoDB using the SQL/MED wrapper [15]. In [2], a SQL engine based on Apache Derby is integrated on top of HBase. The system supports indexes and joins in the relational engine even though they are not supported by HBase. There has also been work mapping XQuery to query NoSQL stores [16]. The Partiqle system [4] part of the CloudDB project [17] supports transactional SQL querying over key-value stores.

NoSQL and relational systems will likely co-exist for some time, and it is valuable to query them simultaneously. Considerable prior work went into the integration of relational systems using schema matching techniques [18], [19] and integration using view and mediators [20]. The same concepts can be applied to NoSQL systems which appear as restricted relational sources to a mediator and integration system. Our prior work on relational integration [21] has been extended to support the querying, integration, and virtualization of both relational and NoSQL systems.

[7]http://pivotalhd.docs.gopivotal.com/getting-started/hawq.html
[8]http://hadapt.com
[9]https://github.com/forcedotcom/phoenix

## III. Use Cases and Motivation

There are three primary reasons for supporting SQL querying of NoSQL systems:

- SQL is a declarative language that allows descriptive queries while hiding implementation and query execution details.
- SQL is a standardized language allowing portability between systems and leveraging a massive existing knowledge base of database developers.
- Supporting SQL allows a NoSQL system to seamlessly interact with other enterprise systems that use SQL and JDBC/ODBC without requiring changes.

The key benefit of supporting SQL is to allow for system portability. NoSQL developers accessing a database must use a custom API and query language. Although queries on key-value stores are simple, database systems are part of a larger enterprise ecosystem and standards facilitate intercommunication. Numerous applications such as query and reporting software access databases and rely on standards such as SQL and ODBC/JDBC. These application vendors are forced to write custom interfaces for their software. This results in slower adoption of NoSQL technologies as vendors will only support systems that get to a certain adoption scale.

There are also many benefits to a virtualization system that has a relational query processor that can execute portions of SQL queries that are not executable by a NoSQL system:

- Allows support for operations not executable by the system such as joins and advanced filtering.
- Isolates users and programmers from writing complicated data manipulation code to implement operators that are not supported by the system.
- Allows for data movement and querying between NoSQL and relational systems.

## IV. Architecture

The architecture of the Unity system is in Figure 1. Unity consists of a SQL query parser that converts a SQL query into a parse tree and validates the query. A query translator converts the parse tree into a relational operator tree consisting of selection, projection, grouping, and join operators. A query optimizer determines join ordering and portions of the query plan to execute on each data source. The execution system interacts with the data sources to submit queries and retrieve results and then perform any additional operations. Each of these key components are discussed in the following sections.

### A. SQL Query Parser

The SQL query parser implemented using JavaCC supports standard SQL-92 syntax for SELECT, INSERT, UPDATE, and DELETE statements including inner and outer joins, delimited identifiers, subqueries, and GROUP BY, ORDER BY, and HAVING. The parser validates for correct SQL syntax and throws exceptions for improper syntax. The output of the parsing step is a parse tree representing the SQL query.

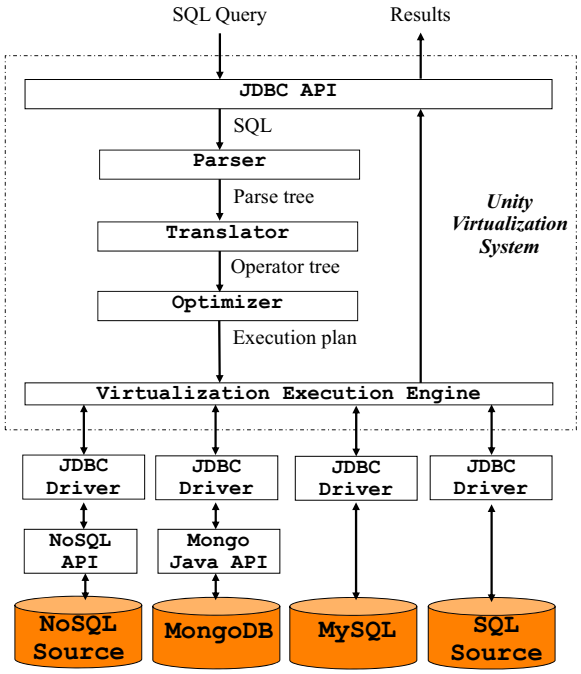


Fig. 1. Unity Architecture

### B. Query Translator

The query translator takes a parse tree and converts it into a relational operator tree. During this conversion, it also validates field and table names provided against the relational schema for the source. For NoSQL systems that do not have a schema, the translator does not validate identifiers and passes them directly to the executor. Invalid identifiers are detected during query execution. Note that when a NoSQL source is accessed in a query with another relational source, the NoSQL source will have a schema defined for it.

### C. Schema Generator

For NoSQL systems such as MongoDB that do not have a schema, the system will optionally generate a schema to describe the data. For MongoDB, this is done by sampling the data in each collection and creating the most general relational model that covers the data. For instance, if the first item in a collection has fields $(a, b)$, the $N$-th item has fields $(a, d, e)$, and the last item has fields $(b, d, f)$, then the relational schema for the collection will be $(a, b, d, e, f)$. A nested attribute such as $c$ that is contained in attribute $a$ is referenced as $a.c$.

The number of items sampled from a collection can be configured. Once defined, a schema can be optionally stored in a MongoDB collection so that it does not have to be re-generated each time.

### D. Query Optimizer

The query optimizer's goal is to perform as much of the query on the individual data sources, following the general

rule of "moving the computation to the data" to achieve better query performance. For a query involving a single relational source, typically the entire query plan can be executed on the relational system. For NoSQL sources, a SQL query may not be executable in its entirety. For instance key-value stores can only perform simple insertions and row selections by key. Document-value stores can perform queries with filters and some ordering. Subqueries, joins, and grouping/aggregation cannot be typically done by NoSQL systems.

For queries involving multiple data sources, the query optimizer must separate out the operators that can be done by each source and push down as much of the computation as possible. Then, it plans the optimal ordering for executing the joins to combine data retrieved from the sources.

There are several techniques used in the optimization:

- Push-down filters - A selection operator is pushed down for execution on a source if possible.
- Join ordering - Join ordering is determined using a cost-based optimizer. The optimizer uses costs that are higher when the join is executed across systems rather than within a data source.
- Push-down, staged joins - A staged join across two systems will retrieve the rows from one join result and use it to modify the query sent to the other source.

Push-down, staged joins are especially powerful. Typically, if a join involves two sources, the data from each source is extracted in parallel using separate threads and then joined in the virtualization engine using hash join. In most cases, this maximizes performance and minimizes data movement. In cases when the number of rows coming back from a source is much smaller than the other, a staged join will be more efficient. The staged join is like a semi-join. The virtualization engine requests and receives rows from the data source that produces the smaller result. It then modifies the query sent down to the other source to have a filter on the join key. The second source will perform the join when it executes its query and only return rows that will be in the join result. The virtualization engine then merges any attributes from the first source query back into the returned rows.

The other form of push-down join pushes the entire data set from one source to the other source to have it execute the join. For relational systems, a temporary table is created. This option is selected when the data being joined is very large and the virtualization engine may be slower than leveraging the parallel computing resources of a source.

During optimization, the optimizer tries to maximize the size of a query plan sent to each source. An operator cannot be sent to a source if it does not support the operator or it does not support some function used by the operator. Those operators are executed by the virtualization system.

### E. Execution Engine

The virtualization engine has implementations of all relational operators including projection, selection, join (nested-loop, sort-merge, hash), sorting, and grouping. These operator implementations are used in query plans that span multiple sources or contain operators that cannot be executed by the underlying sources. The optimizer will have grouped subtrees in the query plan for each source. A root of a subtree can be converted into a SQL query and executed on the source. For relational systems, a SQL query is built (with the proper dialect), and submitted to the source using JDBC. Results are retrieved using a JDBC ResultSet. For NoSQL systems, a JDBC wrapper interface is constructed around the sources' custom API. This wrapper API will accept a SQL query and use the virtualization engine to parse and convert it.

### F. MongoDB SQL/JDBC

For MongoDB, we constructed a JDBC driver that accepts SQL queries. It uses the SQL parser to produce a parse tree and relational operator tree. The converter then traverses the relational operator tree and maps the operator to a MongoDB API call or throws an exception if it cannot be executed.

### G. Function and Dialect Translator

The virtualization engine has a database of mappings between SQL dialects. Key syntax components of SQL and common functions are identified. For each data source, a mapping is created for each function and SQL dialect feature explaining how it is supported for a database. With no mapping, the syntax is assumed to not be supported.

For example, tables and fields can be aliased using "AS". Unity will accept queries that use AS directly or implicitly (as it is optional). Consider a field rename like "myField AS newName". The renaming will be translated and represented in the operator tree. When the operator is converted for a particular database, the translation database is queried for "AS" with a database dialect id given. The dialect id allows looking up an entry for how to convert AS for that data source. Some always require AS while others do not. This feature is also used to support translation of functions.

### H. Example

Consider a customer and order database:

```
Customer(cid,cname,addr.street,addr.city)
Orders(oid,cid,odate,total)
```

Assume that customer information is stored in a MongoDB collection and a sample customer object is:

```
{"cid":1, "cname":"Fred", "address":
    {"street":"Main", "city":"New York"} }
```

The orders information is stored in a table in MySQL and a sample row is `(100,1,'2013-11-10',35.45)`.
This query returns the total of all orders for customer 'Fred' since January 1st, 2013:

```
SELECT SUM(total) as totalAmount
FROM mongo.customer C
 INNER JOIN mysql.orders O ON C.cid=O.cid
WHERE C.cname = 'Fred'
 AND O.odate > '2013-01-01'
```

Tables are prefixed with the source name. For simplicity, we have used the names "mysql" and "mongo". This query would be optimized into the execution plan in Figure 2.

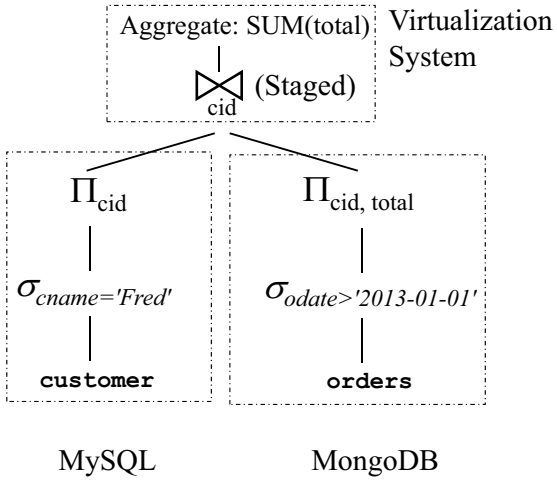


Fig. 2. Query Execution Plan

The filters are pushed down as each one only applies to a single source. The join and grouping is executed in the virtualization layer. This join is performed using a staged join as the number of results returned from Mongo is small which allows for the MySQL query to be modified to return fewer results. The SQL queries executed by each source are:

**Mongo:**

```
SELECT cid FROM customer
WHERE cname = 'Fred'
```

**Mongo JSON query format:**

```
db.customer.find({"cname":"Fred"},
                 {"cid":1})
```

**MySQL:**

```
SELECT cid, total
FROM orders
WHERE odate > '2013-01-01' and cid IN (1)
```

The virtualization system adapts to the properties of the sources. If both of these tables were in MySQL, then the entire query would be executed in MySQL since it supports all operators. If they were stored as MongoDB collections, then two source queries would be executed on MongoDB and the join would be performed at the virtualization layer.

## V. Experimental Results

Unity was evaluated for its performance and overhead. MongoDB was run on an Intel Xeon 2.4 GHz server with 24 GB memory. The client PC used for benchmarking was an Intel PC with 2.8 GHz processor and 8 GB memory.

The performance of the SQL to NoSQL translation was tested using an orders database consisting of 15,000 records. Queries and updates were performed using both the MongoDB Java API and the Unity virtualization engine. The experiments performed 10,000 operations each for SELECT on key, INSERT, DELETE, and UPDATE. SELECT (scan) involved randomly finding a starting record and scanning the next 100 records. 1000 scan operations were performed. The results in Figure 3 are the average of 3 runs.

| | **Mongo** | **Unity** | **% Diff.** |
|---|---|---|---|
| SELECT (key) | 99.4 | 101.1 | 2% |
| SELECT (scan) | 25.1 | 28.0 | 12% |
| INSERT | 11.1 | 14.3 | 30% |
| DELETE | 109.8 | 111.7 | 2% |
| UPDATE | 109.5 | 123.1 | 12% |

Fig. 3. Query Execution Performance (in seconds)

The Unity virtualization driver for most operations has less than 15% overhead. Some overhead is expected as Unity must translate SQL to the Mongo API while also supporting the more complex JDBC API. The only exception is INSERT where the overhead is higher. This is primarily because the insertion time is so low that the overhead is more of a factor. As the operations take more time for the server to process, the overhead becomes a smaller percentage. This means that larger data sets with more users would show better performance as the overhead is a fixed-cost independent of the query size, so the relative overhead decreases as the query time increases.

The performance of cross-database joins was also tested using two databases: MySQL and MongoDB. The join combines a customer table with 1500 records with an orders table containing 15,000 records. The time to read the base relations from Mongo is 0.9 seconds and from MySQL is 1.5 seconds. MySQL takes 2.7 seconds to produce the join result. Unity can join the data where both tables come from Mongo in 1.1 seconds, and where one table comes from Mongo and the other from MySQL in 1.7 seconds. The Unity virtualization engine has comparable performance in the test query compared to executing the query entirely using MySQL. MongoDB cannot execute joins, so the virtualization engine provides an efficient way to join collections in MongoDB.

## VI. Conclusion

Unity is an integration and virtualization system allowing SQL queries over both relational and NoSQL systems. The virtualization layer allows translating SQL queries to NoSQL APIs and automatically executes operations not supported by NoSQL systems. Unity allows NoSQL systems to seamlessly interact with relational databases and enterprise reporting applications. Experimental results show minimal overhead in the SQL translation process. The Unity system has been commercially released as UnityJDBC[10] and the translator for MongoDB packaged as a JDBC driver[11].

[10]http://www.unityjdbc.com

[11]http://www.unityjdbc.com/mongojdbc

Future work involves benchmarking the performance of other supported NoSQL systems such as Cassandra. We are also working on parallelizing the virtualization engine for a cluster environment.

## REFERENCES


[1] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform Access to Non-relational Database Systems: The SOS Platform," in *CAiSE*, 2012, pp. 160–174.

[2] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira, "An Effective Scalable SQL Engine for NoSQL Databases," in *Distributed Applications and Interoperable Systems*. Springer, 2013, pp. 155–168.

[3] A. Calil and R. dos Santos Mello, "SimpleSQL: A Relational Layer for SimpleDB," in *ADBIS*, 2012, pp. 99–110.

[4] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacigümüs, "Partiqle: an elastic SQL engine over key-value stores," in *SIGMOD Conference*, 2012, pp. 629–632.

[5] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[6] "ODBC," in *Encyclopedia of Database Systems*, 2009, p. 1947.

[7] "JDBC," in *Encyclopedia of Database Systems*, 2009, p. 1580.

[8] W. Kim, *Introduction to object-oriented databases*. MIT press Cambridge, MA, 1990, vol. 90.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

[10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "XQuery 1.0: An XML query language," *W3C working draft*, vol. 12, 2003.

[11] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2010.

[12] M. Stonebraker, "SQL databases v. NoSQL databases," *Commun. ACM*, vol. 53, no. 4, pp. 10–11, 2010.

[13] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the Elephants Handle the NoSQL Onslaught?" *PVLDB*, vol. 5, no. 12, pp. 1712–1723, 2012.

[14] J. Roijackers and G. H. L. Fletcher, "On Bridging Relational and Document-Centric Data Stores," in *BNCOD*, 2013, pp. 135–148.

[15] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz, "SQL/MED: A Status Report," *SIGMOD Rec.*, vol. 31, no. 3, pp. 81–89, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/601858.601877

[16] H. Valer, C. Sauer, and T. Härder, "XQuery processing over NoSQL stores," in *Grundlagen von Datenbanken*, 2013, pp. 75–80.

[17] H. Hacigümüs, J. Tatemura, W.-P. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour, "CloudDB: One Size Fits All Revived," in *SERVICES*, 2010, pp. 148–149.

[18] P. A. Bernstein, J. Madhavan, and E. Rahm, "Generic Schema Matching, Ten Years Later," *PVLDB*, vol. 4, no. 11, pp. 695–701, 2011.

[19] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001.

[20] A. Doan, A. Y. Halevy, and Z. G. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012.

[21] T. Mason and R. Lawrence, "Dynamic Database Integration in a JDBC Driver," in *ICEIS*, 2005, pp. 326–333.