Aggregate Computations over Uncertain and Probabilistic Data in TRIO System

Adam Shafi Shaik¹, Sudha Madhuri. M²

¹Pursuing M.Tech(CS) from Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur., Affiliated to JNTUK, Kakinada, A.P., India.

²Asst. Professor, Department of Computer Science Engineering, Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur., Affiliated to JNTUK, Kakinada, A.P., India.

Abstract - Trio is a new kind of database system that supports data, uncertainty, and lineage in a fully integrated manner. The first Trio prototype, dubbed Trio-One, is built on top of a conventional DBMS using data and query translation techniques together with a small number of stored procedures. Uncertain and probabilistic data are inherent in many important applications. Recently, considerable research efforts have been put into the field of managing uncertain data. This paper describes Trio-One’s translation scheme and system architecture, showing how it efficiently and easily supports the Trio data model and query language and also shows how aggregation is performed on uncertain and probabilistic data in TRIO system.

Keywords - Aggregation, Uncertain data, TRIO system, Probabilistic data.

1. Introduction

Managing uncertain data has been studied ever since the eighties last century from the database society. With the emergence of many recent important and novel applications involving uncertain data, there has been a great deal of research attention dedicated to this field. The applications include data cleaning, data integration, information extraction, sensor networks, economic decision making, market surveillance, trend prediction, moving object management, etc. Uncertainty is inherent in such applications due to various factors such as data randomness and incompleteness, limitation of equipment, and delay or loss in data transfer.

Managing uncertain data is not well supported by conventional database systems [4]. A number of technical issues in traditional databases have been reinvestigated recently under uncertain semantics, including modeling uncertainty, query evaluation, indexing, query processing against relational and spatial uncertain data. Many important results have been obtained in system and theory and many systems have been developed and implemented to support uncertain data management. The most important one is the TRIO system.

Trio is developed by Standford InfoLab [1], [2]. As a database system, it not only tackles modelling and analyzing data but also the accuracy and lineage of data. Trio is developed based on data model ULDB. It is implemented on the top of traditional relational DBMS (PostgreSQL). Query language in Trio is an extension of SQL. TrioQL. TrioQL handles queries, as well as accuracy and lineage of data. Figure 1 illustrates the system architecture of Trio [3]. The Trio API accepts TrioQL as well as regular SQL queries from client and translates them into standard SQL queries; in the relational DBMS, data tables are encoded, namely, are integrated with uncertainty information such as confidence. Trio Stored Procedures handles confidence and lineage information.

Fig. 1. Trio System Architecture
2. The TRIO System

We now briefly describe the architecture and features of the demonstrated Trio prototype. More details can be found in [2].

2.1 Software Architecture

The initial Trio system is built entirely on top of a conventional relational DBMS: ULDBs are represented in relational tables, and TriQL queries and commands are rewritten automatically into SQL commands evaluated against the representation. The core system is implemented in Python and presents a simple API that extends the standard Python DB 2.0 API for database access (Python’s analog of JDBC). The Trio API supports TriQL queries instead of SQL, query results are cursors enumerating x-tuple objects instead of regular tuples, and x-tuple objects provide programmatic access to their alternatives, including confidences and lineage. Using the Trio API, we build a generic command-line interactive client similar to that provided by most DBMS’s, and a full featured graphical user interface, TrioExplorer.

ULDBs and TriQL queries are demonstrated through TrioExplorer, a generic graphical user interface built on top of Trio’s API. TrioExplorer lets the user connect to a ULDB, browse its schema and schema-level lineage structure, browse data in x-relations, ask TriQL queries and visualize their results, and navigate lineages of stored x-relations and query results.

2.2 Functionality

The user may create a ULDB x-relation $R$ with any standard relational schema, with or without confidence values. This x-relation is represented as a conventional table storing tuples in $R$’s schema augmented with aid’s (globally unique alternative identifiers) and xid’s (x-tuple identifiers that encode which alternatives belong to the same x-tuple). When a TriQL query creates a derived x-relation $S$, in addition to creating a table for $S$ as described above, an additional table ‘lin:S’ is created to store the lineage of data in $S$. Tuples in the lineage table contain three attributes: the aid for an alternative in $S$, and the table-name and aid for an alternative in $S$’s lineage. Trio also maintains a catalog containing Trio-specific metadata, such as schema-level lineage information and which x-relations contain confidence values.

TriQL queries are rewritten automatically into SQL commands over the representation just described. Query processing proceeds in two phases. In the first phase, a single SQL query generates a table (call it $T$) containing all information needed to construct the query result. In the second phase, table $T$ is post-processed to correctly group alternatives into the x-tuples in the result, and to construct the lineage table for the result. This second phase offers two options: It can produce the final result as a stored xrelation, or it can expose a cursor, in which case result xtuples and their lineages are assembled as the application iterates through the cursor. Result confidence values are either computed immediately as part of the second phase, or on-demand when requested by a user or by a subsequent TriQL query with a conf() predicate.

3. TriQL: Query language

We present the ULDB model and TriQL through examples using a highly simplified “crime-solver” application. Tables Drives(person, car) and Saw(witness, car) capture (possibly uncertain) driver information and crime-vehicle sightings, respectively.

ULDBs extend the standard SQL relational model with four new features:
1. *alternatives*, representing uncertainty about the contents of a tuple
2. *maybe* (“?”) annotations, representing uncertainty about the presence of a tuple
3. numerical *confidence* values optionally attached to alternatives and “?”
4. *lineage*, connecting tuple alternatives to other alternatives from which they were derived.

We now introduce *TriQL*, Trio’s SQL-based query language. Except for built-in functions and predicates for querying confidence values and lineage, TriQL uses the same syntax as SQL. As a first example, the join query with its result stored in table Suspects would be written in TriQL simply as:

```
TriQL> CREATE TABLE Suspects AS
TriQL> SELECT person
TriQL> FROM Saw, Drives
TriQL> WHERE Saw.car = Drives.car
```

In addition to modifying SQL semantics for ULDBs, TriQL adds a number of new constructs for querying and manipulating both uncertainty and lineage. A comprehensive specification for TriQL’s query and update language appears in [5]. In the remainder of this section we use examples to illustrate TriQL semantics and functionality, and how TriQL queries are rewritten.
automatically into standard SQL over the relationally-encoded ULDB data.

3.1 Basic Rewriting Scheme

Consider the Suspects query shown above, first in its transient form (i.e., without CREATE TABLE). The Trio Python layer translates the TriQL query into the following SQL query, sends it to the underlying DBMS, and opens a cursor on the result:

```sql
SELECT Drives.person, Saw.aid, Drives.aid, Saw.xid, Drives.xid, (Saw.num * Drives.num) AS num
FROM Saw, Drives
WHERE Saw.car = Drives.car
ORDER BY Saw.xid, Drives.xid
```

Let $Tfetch$ denote a cursor call to the Trio API for the original TriQL query, and let $Sfetch$ denote a cursor call to the underlying DBMS for the translated SQL query. Each call to $Tfetch$ must return a complete x-tuple, which may entail several calls to $Sfetch$: Each tuple returned from $Sfetch$ on the SQL query corresponds to one alternative in the TriQL query result, and the set of alternatives with the same returned Saw.xid and Drives.xid pair comprise a single result x-tuple. (The TriQL operational join semantics presented in [3] makes this property very clear.) Thus, on $Tfetch$, Trio collects all SQL result tuples for a single Saw.xid/Drives.xid pair (enabled by the ORDER BY clause in the SQL query), generates a new xid and new aid’s, and constructs and returns the result x-tuple.

Note that the underlying SQL query also returns the aid’s from Saw and Drives. These values (together with the table names) comprise the lineage for the alternatives in the result x-tuple. As mentioned earlier, the num field is used to encode the presence or absence of “?”. Finally, since result confidence values for joins are not computed until they are explicitly requested, $Tfetch$ initially returns NULL confidence for all alternatives, whether or not the query result logically contains confidence values. For the stored (CREATE TABLE) version of the query, Trio first issues DDL commands to create new tables for the query result and its lineage. Trio then executes the same SQL query shown above, except instead of constructing and returning x-tuples one at a time, the system directly inserts the new alternatives and their lineage into the result and lineage tables, already in their encoded form. All processing occurs within an SPI stored procedure on the database server, thus avoiding unnecessary roundtrips between the Python module and the underlying DBMS.

3.2 Aggregation

TriQL supports standard SQL grouping and aggregation. Consider the following query:

```triql
SELECT car, count(*)
FROM Drives GROUP BY car
```

The query result appears fairly straightforward for our very simple example, although notice that tuple 91 is the result of merging two duplicate alternatives.

<table>
<thead>
<tr>
<th>ID</th>
<th>(car, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>(Mazda, 1)</td>
</tr>
<tr>
<td>92</td>
<td>(Honda, 1)</td>
</tr>
</tbody>
</table>

In general, aggregation can be an exponential operation in ULDBs (and in other data models for uncertainty). Thus, TriQL includes built-in approximate aggregation functions, including low and high bounds for the aggregate result, and expected values that take confidence into account. For example, the following query returns expected values for the number of occurrences of each type of car in Drives.

```triql
SELECT car, ecount(*)
FROM Drives GROUP BY car
```

The result on our example Drives table with confidence values is (Mazda, 1.0), (Honda, 1.8).

TriQL supports 20 different aggregation functions: four versions (full, low, high, and expected) for each of the five standard functions (count, min, max, sum, avg). (Distinct versions of the aggregation functions currently are not supported.) All of the full functions and some of the approximations unfortunately cannot be translated to SQL queries over the encoded data, and thus are implemented as algorithmic stored procedures. Furthermore, several of the low/high bounds and one of the expected values are themselves approximations to the tightest bound or value, because finding the exact answer based on possible-instances can be extremely expensive. (We expect the approximations to do well in practice, but details are far beyond the scope of this description.) Many of the approximate functions can be implemented exactly and translated very easily. For example, for the ecount TriQL example above, the SQL query over the encoded data is simply:
SQL> SELECT car, sum(conf)
FROM Drives
GROUP BY car

Note that with approximate aggregation, query results are regular tables and not ULDB tables: they do not include alternatives, “?” confidence values, or lineage.

4. Conclusion

Almost every problem in conventional databases needs to be reinvestigated under uncertain semantics since the uncertain nature poses great and unique technical challenges. This paper describes Trio-One’s translation scheme and system architecture, showing how it efficiently and easily supports the Trio data model and query language and also shows how aggregation is performed on uncertain and probabilistic data in TRIO system.

References


