Data Integration in Vector (Vertically Partitioned) Databases

Peter Buneman Jon G. Riecke Eric Sandler Vladimir Seroff
University of Edinburgh * Aleri Inc. †

Abstract

Data integration requires not only the absorption of data, but the transformation of that data. For instance, manufacturing companies must combine customer order information for ordering supplies as well as for financial planning; large banks may need to produce consolidated profit-and-loss statements daily, or even more frequently, to manage liquidity and risk. The data may be spread across many heterogeneous databases located in different countries, with different standards of cleanliness, described in different currencies or different units of measure. The transformation rules may also change over time when, for instance, companies merge or split, or new rules of accounting are imposed. Clearly it is best if the data integration tool is flexible enough not only to accomodate new sources of data, but also to change the rules of combining that data. This paper describes one data integration tool, Aleri Inc.'s Modeler. We focus primarily on one aspect of Modeler: the use of vectorization both as an implementation technique and as the fundamental unit of computation. Vectorization improves both the efficiency and the ease-of-use of Modeler.

1 Vectorization

1.1 Vertical Partitioning

The widely accepted storage technique for relational databases is to store each tuple contiguously. An alternative, which is almost as old as relational databases [4], is *vertical partitioning*, in which one stores the columns contiguously. The benefit is that queries that only involve a small number of columns require less i/o. Moreover, there are dramatic performance improvements to be made if the columns can compactly represented and manipulated as vectors in main memory.

The idea of vertical partitioning has re-emerged for implementing object-oriented databases [3, 2] and to speed up transfer between main memory caches and disk in relational databases [1]. It has also been used commercially in Sybase IQ and recently by Aleri Inc. where it is combined with vector processing language technology and has been successfully deployed in a number of financial database applications to support data integration and OLAP. We shall use the term "vectorization" for that form of vertical partitioning which stores the columns of a table as vectors and, in particular, preserves the *order* in each column, which is an added bonus for applications in which the vertical order of a table is needed; for example the table may represent a time-series

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*http://www.informatics.ed.ac.uk/people/staff/Peter_Buneman.html
†http://www.Aleri.com

thereby departing from the relational (SQL) model in which a table is defined as a *set* (*multiset*) of tuples. There are, of course, a few drawbacks to vectorization. Row deletion is prohibitively expensive if naively implemented. Also, there is a sense in which vertical partitioning is used in conventional database implementations: an index typically stores information about one column or a small subset of columns contiguously, and it is occasionally possible to answer a query on the basis of an index alone.

1.2 Vector Languages

The language APL [5] is based on a number of array manipulation primitives. It shares with relational algebra the principle that languages based on "bulk" operations are more concise and lend themselves to optimization both through efficient implementation of those operations and through code rewriting.

VCL, the main implementation language used by Aleri Inc., is one of a number of derivatives of APL in use today. Apart from the superficial, but useful distinction that VCL uses a conventional character set, VCL is a higher-order language in the sense that functions are first-class values, and functions can take other functions as arguments and return other functions as results.

For readers who have not seen APL or its derivatives, it may help to give the flavor of VCL through a few examples on vectors (one-dimensional arrays). The operator "+" is overloaded to add a scalar to an array:

$$5 + (\,4\ 7\ 6\,) \models\ 9\ 12\ 11$$

(we shall use the symbol $\models$ to mean "evaluates to"). This operator is further overloaded to work pointwise on vectors of equal length:

$$(\,3\ 2\ 1\,) + (\,4\ 7\ 6\,) \models\ 7\ 9\ 7$$

Indexing is also overloaded in that one can index one vector by another to produce a vector:

$$(\,5\ 7\ 2\ 4\ 1\,)[\,3\ 0\ 4\,] \models\ 4\ 5\ 1$$

An operation that produces an indexing vector from a vector is "&" which operates on a boolean (0 or 1) vector to produce the positions in which that vector is 1

$$\&(\,0\ 0\ 1\ 0\ 1\ 1\,) \models\ 2\ 4\ 5$$

Putting these operations together, suppose we have a relational table R with attributes A and B and we wish to evaluate a simple select-project query SELECT A FROM R WHERE B = 123. In a vector representation of this table we label the columns R.A and R.B and evaluate the expression

$$\text{R.A}[\ \&(\text{R.B} = 123)] \qquad \text{(a)}$$

To explain what is happening here, the boolean operator "=" is, like "+", overloaded to produce a boolean vector. From this an indexing vector is produced by &, which is used to index into R.A. Note that, like SQL, this operation does not eliminate duplicates, but there is a further VCL operation that produces unique values within a vector.

1.3 Using Vectorization to Implement Relational Databases

Example (a) illustrates the overwelming advantage of vectorization, namely that one only needs to "read" the vectors R.A and R.B in order to evaluate the query. Contrast this with the conventional implementation of a relational database system in which (unless one could capitalize on information stored in an index) one would read all the tuples. This, taken with the fact that a relational database system typically does not fill disk pages to capacity, leads to a dramatic reduction in i/o.

At the same time, we do not want to take the VCL evaluation of a query literally. Example (a) appears to call for the creation of two intermediate vectors, but this is clearly not neccessary. Another case in point is join evaluation. If one represents columns of a table as vectors, one is tempted to think that replacing a foreign key by an index into the table referenced by the foreign key will speed up a join. However consider the problem of evaluating $A[V]$ where both A and V are too large for main memory. The naive evaluation requires approximately $|V|$ reads of A; but we know we can do much better using a hash or sorted join. Such a join may be useful even in main memory evaluation where it could be used to reduce transfer into caches. The point to be made is that use of *both* vector processing techniques and database technology is needed for implementation of vectorized databases.

In addition to implementing a query engine for vectorized databases, Aleri has developed (a) transaction processing software (b) a declarative system, Modeler, for data integration and (c) efficient view maintenance technology that depends both on the vector organization of the database and on (a) and (b). In this paper we shall focus on (b) and remark briefly on (c).

2 Data Integration using Modeler

Modeler is a declarative language for data integration, view/query definition and OLAP. It is based on the use of rules and structural assertions that guide the formation of views. In the following account we use a surface syntax that is somewhat different than that used in Modeler, which is partly controlled by a graphical interface. Let us start with a simple example, in which we are given the source tables:

order:

oid*	cid	date
17	Jones	07/21/01
18	Smith	09/22/01

olist:

oid*	pid*	qty	priceq
17	A4	100	145.00
17	B7	15	32.15
18	A4	10	16.00

The order table gives order details: customer and date; the olist table gives the parts associated with each order, the quantity, and the quoted price for that order. The keys for each table are flagged with an asterisk; oid and pid form a joint key for the olist table.

The simplest form of "rule" is based on one table:

olist.unitprice := olist.priceq / olist.qty

This provides the price per part associated with each order by augmenting the olist table with a new column:

olist:

oid*	pid*	qty	priceq	*unitprice*
17	A4	100	195.00	1.95
17	B7	15	32.15	2.14
18	A4	10	16.00	1.60

We have italicized the unitprice header to indicate that it is a derived column. This operation is unconventional in two ways:

- Because of the vectorized organization of the database, no internal reorganization is required, and the computation only requires access to the priceq and qty columns.
- This is more than a conventional SQL "add column" operation. Updates to either the priceq or qty columns will be immediately reflected in the derived unitprice column. In effect we are changing the olist table so that part of it is the source table and part is a materalized view.

One of the most important features of vectorized databases is the ability to do efficient and general materialized view maintenance. By contrast the construction of materialized views in conventional database systems is subject to a host of restrictions, the most serious being that they cannot be composed. In Modeler, even though one can declare new tables and think of a view as a new table, the fundamental operation is creating a new column or vector.

Continuing with our example, suppose we want to extract information from the combined tables. Our first task is to say how the tables are to be combined. This is performed by a *connecting* rule:

connect(olist.oid = order.oid)

Although this looks like a join instruction, its function is not to perform an immediate join but to connect the tables for the purpose of deriving further data – by means of a join – at a later stage. For example:

olist.date	:=	order.date
order.total	:=	sum(olist.priceq)

The first of these, rather trivially, adds date information to the olist table; the second performs an aggregation. The key structure of the two tables and the form of connect declaration tells us that there is a *many-one* relationship between the two tables. Thus aggregation is appropriate in one direction but not in the other.

We have used two kinds of rules in this process: *derivation rules* which generate columns and *connecting rules* which describe how tables are to be connected. Either kind of rule may be accompanied by a condition such as olist.date $>$ 08/17/02 which, in the case of a connecting rule, selects a subset of the data for consideration.

So far our examples have only added data to existing columns. In general we shall want to construct new tables which, in conventional database parlance are *materialized views*. To do this we first declare the structure of the new table:

part(pid*, total)

Any table declaration must include a key field. We also make the connecting declaration connect(part.pid = olist.pid). This table may now be populated by the derivation rules:

part.pid	:=	olist.pid
part.total	:=	sum(olist.qty)

This generates the derived table

part:

*pid**	*total*
A4	110
B7	15

Of these derivation rules, the derivation of the key, pid is pivotal. It is evaluated prior to any other rule for columns in the part table. Moreover, it generates a set of distinct values, because pid is a key. The non-key columns in the table can be constructed by the process outlined above. In fact, the only thing we need to do is to establish a new table is to provide a key and a derivation rule for that key. We can continue to define further columns independently, e.g.

part.aveprice	:=	sum(olist.qty $\star$ olist.priceq) / part.total
part.delay	:=	min(order.date)

Note that the first of these rules uses columns from different tables, and the second derives data via two connecting rules.

At this point we should discuss under what circumstances a set of rules is meaningful. Suppose the connecting rule is connect$(a_1 = a_2)$ where a_1, a_2 are attributes of R_1, R_2 respectively. If a_1 and a_2 are both keys, the connection induces a *one-one* relationship between R_1 and R_2; if one of them is a key, the relationship is *one-many*; and if neither is a key the relationship is *many-many*. This analysis can be extended, in certain cases, to more general connecting rules of the form connect$(e_1 = e_2)$ where e_1 and e_2 are respectively expressions on the attributes of R_1 and R_2. For example we can connect on the basis of multiple attributes, connect$((a_1, b_1) = (a_2, b_2))$. It is also possible to extend connecting assertions to certain other expressions, such as connect(r.agegroup = trunc(s.age / 10)).

From this analysis we obtain a graph, whose nodes are the tables in the database and whose edges are marked to indicate the kind of relationship (many-one, one-one, etc) to indicate the kind of relatyionship. We can reasonably refer to this graph as the *schema*, for it has a close connection to the entity-relationship diagram

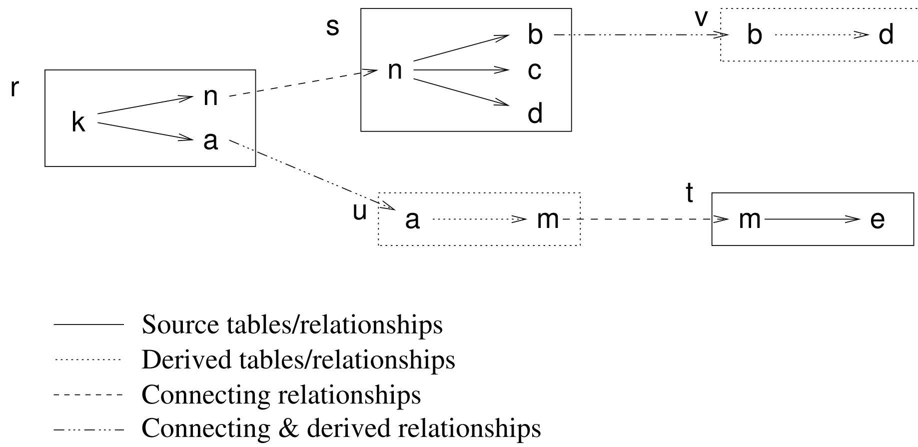

Figure 1: A schema graph generated by rules

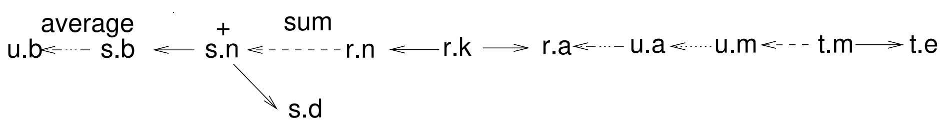

Figure 2: The tree used in the computation of v.d

that one might design in order to describe this integration process. Our first requirement is that the schema, viewed as an undirected graph, should be a *tree*. This guarantees that there is at most one path between any two attributes, which is needed to ensure that the derivation of one attribute from other attributes is unambiguous.

Our second requirement has to do with the derivation rules. It ensures that there is no circularity in the derivation of attributes and that there are enough rules to ensure that every attribute gets derived – directly or indirectly – from source data. The schema is also used in the compilation of rules into executable code. The details are not given here, but the following example illustrates the issues involved.

Source tables:	r(k*,n,a), s(n*,b,c,d), t(m*,e)
Derived tables:	u(a*,m), v(b*,d)
Connecting rules:	connect(r.n=s.n), connect(t.m=u.m), connect(r.a=u.a), connect(s.b=v.b)
Derivation rules:	u.a:=r.a, u.m:=min(s.c), v.b:=s.b, v.d:=average(s.d + sum(t.e))

Figure 1 shows the schema that is derived from this set of rules. Each box describes one table, the arrows within each box show the many-one relationship between the key and the other attributes. A corresponding entity-relationship (ER) diagram can be derived from this figure simply by replacing each edge between tables with an ER relationship node and creating the obvious ER attribute nodes.

Figure 2 shows the tree of dependencies used in the computation of the column v.d. Computing this column is tantamount to computing the values of a function on the key v.b. The details are not given here, but to sketch what happens: First, the expressions s.d and sum(t.e) "converge" at s.n. The arrows between s.n and t.e give rise to a many-many relationship between these columns, and with a group-by and sum one can obtain a further derived column of r. Now add this column to s.d to obtain another derived column of r. Between this column and r.b we have a many-many relationship and again we can perform a group-by and an average.

It is also worth observing that, if the purpose of the rules is to derive the v table, there is an equivalent set of rules (corresponding to an equivalent ER diagram) in which the u table has been replaced by an additional column in the r table.

Source tables:	r(k*,n,a), s(n*,b,c,d), t(m*,e)
Derived tables:	v(b*,d)
Connecting rules:	connect(r.n=s.n), connect(t.m=r.m), connect(s.b=v.b)
Derivation rules:	r.m:=min(s.c), v.b:=s.b, v.d:=average(s.d + sum(t.e))

The ability to perform unions is an important part of data integration. In Modeler this is achieved (as in datalog) by providing two derivation rules for keys. Continuing our parts/orders example, suppose there is an additional base table supplies(supplier*,pid*,unitprice) we may add the derivation rule for part:

part.pid := supplies.pid

Other attributes in the derived part table must be changed appropriately, e.g.:

part.total := ifnull(olist.pid) then 0 else sum(olist.qty)

This also illustrates the use of a conditional in a derivation rule. Using conditions and aggregates, it is possible to express set membership, hence negation and subtraction. This enables us to express all operations of the relational algebra.

3 View maintenance in Modeler

We noted before that Modeler does more than "one-shot" data integration; the derived data is updated in response to updates to source tables. This is materialized view maintenance in conventional databases, but in part because of the vector representation, views can be more efficiently maintained, and they are fully compositional.

There are two reasons for the gain in efficiency. The first derives from the fact that the queries involved, even complex aggregate queries, typically involve only a small number of source columns, therefore – as we have already noted – recomputation of queries is inherently more efficient. The second reason is more subtle. Recall that certain updates such as deletion are expensive in vectorized databases, but if we are in an environment in which the majority of updates are modifications we can again capitalize on the vectorized representation. Modifications to a table are not random: in most situations one or two columns in a table are highly volatile, while the remaining columns will remain relatively stable. Thus we can tune the "refresh" algorithms to try to keep memory-resident those vectors that derive from volatile columns.

At the core of the view-maintenance algorithm is an "indices table" that keeps track of what source attribute values can affect derived attribute values. Consider an update to s.d in the example above. If the sum(t.e) vector is materialized, then we need to do one addition and recompute a new average in order to maintain the view v. The indices table keeps track of which elements are needed in the recomputation of the average. Note how, once again, vectorization helps in this process. View maintenance requires the computation of a few vectors rather than updates to a number of tables.

4 Conclusions

We have attempted to describe how a vectorized representation of relational databases is useful in data integration. It is useful at the conceptual level because the fundamental unit of construction is the column; it is useful in implementation because not only the initial computation of derived data, but also refreshing that data in response to modification of the source – materialized view maintenance – is much more efficient.

In basing the declarative formulation of views on relationships (many-one, one-one, etc.) derived from keys, Modeler is close in spirit to the Clio project [6] at IBM. The idea of using an tree-like structure to ensure unique derivation paths without specifying the paths bears some relationship to the acyclic hypergraph schemes of the universal relation assumption [7] – though in that model there was no method of deriving new columns.

Compared with other methods of data integration that exploit high-level knowledge representation formalisms, Modeler may appear somewhat low-level. Nevertheless it is simple, efficient and may serve as supporting technology for a higher-level formalism, should one ever become popular.

Acknowledgements

We are grateful to Renée Miller for her comments on a draft of this paper.

References

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[2] Peter A. Boncz, Annita N. Wilschut, and Martin L. Kersten. Flattening an Object Algebra to Provide Performance. In *International Conference on Data Engineering*, pages 568–577, 1998.

[3] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279. ACM Press, 1985.

[4] Don S. Batory. On Searching Transposed Files. *ACM Transactions on Database Systems*, 4(4):531–544, 1979.

[5] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, 1962.

[6] Renée J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 77–88, Cairo, Egypt, Sept 2000.

[7] Jeffrey D. Ullman. On Kent's "Consequences of Assuming a Universal Relation". *ACM Transactions on Database Systems*, 8(4):637–643, 1983.