Symmetric Relations and Cardinality-Bounded Multisets in Database Systems

Kenneth A. Ross Julia Stoyanovich

Columbia University*
kar@cs.columbia.edu, jds1@cs.columbia.edu

Abstract

In a binary symmetric relationship, A is related to B if and only if B is related to A. Symmetric relationships between k participating entities can be represented as multisets of cardinality k. Cardinality-bounded multisets are natural in several real-world applications. Conventional representations in relational databases suffer from several consistency and performance problems. We argue that the database system itself should provide native support for cardinality-bounded multisets. We provide techniques to be implemented by the database engine that avoid the drawbacks, and allow a schema designer to simply declare a table to be symmetric in certain attributes. We describe a compact data structure, and update methods for the structure. We describe an algebraic symmetric closure operator, and show how it can be moved around in a query plan during query optimization in order to improve performance. We describe indexing methods that allow efficient lookups on the symmetric columns. We show how to perform database normalization in the presence of symmetric relations. We provide techniques for inferring that a view is symmetric. We also describe a syntactic SQL extension that allows the succinct formulation of queries over symmetric relations.

*This research was supported by NSF grants IIS-0120939 and IIS-0121239.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

A relation R is *symmetric* in its first two attributes if $R(x_1, x_2, \ldots, x_n)$ holds if and only if $R(x_2, x_1, \ldots, x_n)$ holds. We call $R(x_2, x_1, \ldots, x_n)$ the *symmetric complement* of $R(x_1, x_2, \ldots, x_n)$. Symmetric relations come up naturally in several contexts when the real-world relationship being modeled is itself symmetric.

Example 1.1 *In a law-enforcement database recording meetings between pairs of individuals under investigation, the "meets" relationship is symmetric.* □

Example 1.2 *Consider a database of web pages. The relationship "X is linked to Y" (by either a forward or backward link) between pairs of web pages is symmetric. This relationship is neither reflexive nor antireflexive, i.e., "X is linked to X" is neither universally true nor universally false. While an underlying relation representing the direction of the links would normally be maintained, a view defining the "is linked to" relation would be useful, allowing the succinct specification of queries involving a sequence of undirected links.* □

Example 1.3 *Views that relate entities sharing a common property, such as pairs of people living in the same city, will generally define a symmetric relation between those entities.* □

Example 1.4 *Example 1.1 can be generalized to allow meetings of up to k people. The k-ary meeting relationship would be symmetric in the sense that if $P = (p_1, \ldots, p_k)$ is in the relationship, then so is any column-permutation of P.* □

Example 1.5 *Consider a database recording what television channel various viewers watch most during the 24 hourly timeslots of the day.*[1] *For performance reasons,*[2] *the database uses a table*

[1] This example is based on a real-world application developed by one of the authors, in which there were actually 96 fifteen-minute slots.

[2] A conventional representation as a set of slots would require a 24-way join to reconstruct V.

$V(ID, ViewDate, C_1, \ldots, C_{24})$ to record the viewer (identified by ID), the date, and the twenty-four channels most watched, one channel for each hour of the day. This table V is not symmetric, because C_i is not interchangeable with C_j: C_i reflects what the viewer was watching at timeslot number i. Nevertheless, there are interesting queries that could be posed for which this semantic difference is unimportant. An example might be "Find viewers who have watched channels 2 and 4, but not channel 5." For these queries, it could be beneficial to treat V as a symmetric relation in order to have access to query plans that are specialized to symmetric relations. □

There is a natural isomorphism between symmetric relationships among k entities, and k-element multisets.[3] We phrase our results in terms of "symmetric relations" to emphasize the column-oriented nature of the data representation in which columns are interchangeable. Nevertheless, our results are equally valid if expressed in terms of "bounded-cardinality multisets".

Sets and multisets have a wide range of uses for representing information in databases. Bounded cardinality multisets would be useful for applications in which there is a natural limit to the size of multisets. This limit could be implicit in the application (e.g., the number of players in a baseball team), or defined as a conservative bound (e.g., the number of children belonging to a parent). We will demonstrate performance advantages for bounded-element multisets compared with conventional relational representations of (unbounded) multisets.

Storing a symmetric relation in a conventional database system can be done in a number of possible ways. Storing the full symmetric relation induces some redundancy in the database: more space is required (up to a factor of $k!$ for k-ary relationships), and integrity constraints need to be enforced to ensure consistency of updates. Updates need to be aware of the symmetry of the table, and to add the various column permutations to all insertions and deletions. Queries need to perform I/O for tuples and their permutations, increasing the time needed for query processing.

Alternatively, a database schema designer could recognize that the relation was symmetric and code database procedures to store only one representative tuple for each group of permuted tuples. A view can then be defined to present the symmetric closure of the stored relation for query processing. The update problem remains, because updates through this view would be ambiguous. Updates to the underlying table would need to be aware of the symmetry, to avoid storing multiple permutations of a tuple, and to perform a deletion correctly. For symmetric relations over k columns, just defining the view (using standard SQL) requires a query of length proportional to $k(k!)$.

[3] A multiset is a set except that duplicates are allowed.

For both of the above proposals, indexed access to an underlying symmetric relationship would require multiple index lookups, one for each symmetric column.

A third alternative is to model a symmetric relation as a set [3] or multiset. Instead of recording both $R(a, b, c, d, e)$ and $R(b, a, c, d, e)$, one could record $R'(q, c, d, e)$, $S(a, q)$, and $S(b, q)$, where q is a new surrogate identifier, and R' and S are new tables. The intuition here is that q represents a multiset, of which a and b are members according to table S. Distinct members of the multiset can be substituted for the first two arguments of R. To represent tuples that are their own symmetric complement, such as $R(a, a, c, d, e)$, one inserts $S(a, q)$ twice. This representation uses slightly more space than the previous proposal, while not resolving the issue of keeping the representation consistent under updates. Further, reconstructing the original symmetric relation requires joins.

We argue that none of these solutions is ideal, and that the database system should be responsible for providing a "symmetric" table type. There are numerous advantages to such a scheme:

1. The database system could choose a compact representation (such as storing one member of each pair of symmetric tuples) and take advantage of this compactness in reducing the amount of I/O required. This representation can be used both for base tables that are identified as symmetric, and for materialized views that can be proven to be symmetric.

2. The database system could go even further, and add a symmetric-closure operator to the query algebra. A query plan over a symmetric relation could then be manipulated using algebraic identities so that the symmetric closure is applied as late as possible. That way, intermediate results will be smaller, and queries will be processed more efficiently.

3. Integrity would be checked by the database system. Single-row updates would be automatically propagated to the other column permutations if necessary. Inconsistencies would be avoided, and schema designers would not have to re-implement special functionality for each symmetric table in the database.

4. The database system could index the multiple columns of a symmetric relation in a single index structure. As a result, only one index traversal is necessary to locate tuples with a given value for some symmetric column.

In this paper, we propose techniques to enable such a "symmetric relation" table type. We provide:

- An underlying abstract data type to store the *kernel* of a symmetric relation, i.e., a particular nonredundant subset of the relation. We show how updates on this data type would be handled by the database system. We describe how relational normalization techniques should take account of symmetric relations during database design. Both normalization and the proposed representation of symmetric relations aim to remove redundancy, so combining these two approaches should be beneficial.

- An extension of the relational algebra with a symmetric closure operator γ. We show how to translate a query over a symmetric relation into a query involving γ applied to the kernel of the relation. We provide algebraic equivalences that allow the rewriting of queries so that work can be saved by applying γ as late as possible.

- A method for inferring when a view is guaranteed to be symmetric. By using this method, the database system has the flexibility to store a materialized view using the more compact representation.

- A syntactic extension to SQL that allows the succinct expression of queries over symmetric relations.

Related Work

Surprisingly, there has been little past work on specialized implementations of symmetric relations (or bounded-cardinality set/multisets) within the database system. The only literature we are aware of that addresses this problem is [3], where database-level implementation is advocated, but specific implementation techniques are not described.

An object-relational database system can provide explicit structures for representing set-valued attributes that are physically embedded in a stored tuple, and can be manipulated directly [6, 7, 8]. For example, Oracle provides an object-relational collection type called a `VARRAY` [1]. `VARRAY`s allow the embedded representation of arrays having a fixed cardinality bound. A database schema designer could use this kind of system to implement the set-based representation of symmetric relations mentioned above, without the need for joins to reconstruct the symmetric relation. Nevertheless, one must give up first normal form and/or use an extended relational database system. Further, the encapsulation of these collection types means that the full set has to be dereferenced for accesses and element updates. For example, it is not possible to index the elements of a `VARRAY`, and so finding rows with `VARRAY`s containing a particular element must be performed using a full table scan.

The expressive power of cardinality-bounded sets has been previously studied in the context of an object-based data model [4, 5].

2 The Kernel

Definition 2.1 *$\gamma_{XY}(R)$ denotes the symmetric closure operator over symmetric attributes X and Y of relation $R(X, Y, Z_1, \ldots, Z_n)$.*[4] *$(x, y, z_1, \ldots, z_n) \in \gamma_{XY}(R)$ if and only if either $(x, y, z_1, \ldots, z_n) \in R$ or $(y, x, z_1, \ldots, z_n) \in R$.* □

If R is symmetric with respect to X and Y, then we aim to determine a minimal relation M such that $R = \gamma_{XY}(M)$. By choosing a minimal M, we can represent R compactly. Several minimal relations M satisfy this constraint. Each such M chooses a particular element from each pair of complementary tuples.

While the choice of minimal relation M does not matter in terms of space consumption, we shall see that certain algebraic equivalences (such as Lemma 3.4 below) hold only if there is a consistent single choice of M for all tables. Thus, we impose a total order (which may be arbitrary) on the domain of X and Y, and insist that the representative tuple chosen has $X \leq Y$ according to this order. The resulting relation is unique, and is denoted by $ker_{XY}(R)$, or just $ker(R)$ when X and Y are clear from context. $ker_{XY}(R) = \sigma_{X \leq Y}(R)$.

We propose that the database stores $ker(R)$ as the internal representation of R. Assuming a set semantics (as opposed to a multiset or bag semantics) for symmetric relations, updates are handled as follows:

```
Insert ( R(X,Y,Z1,...,Zn) )
{
  If (Y<X) then swap(X,Y);
  If (X,Y,Z1,...,Zn) is not in ker(R) then
    append (X,Y,Z1,...,Zn) to ker(R);
}

Delete ( R(X,Y,Z1,...,Zn) )
{
  If (Y<X) then swap(X,Y);
  If (X,Y,Z1,...,Zn) is in ker(R) then
    remove (X,Y,Z1,...,Zn) from ker(R);
}
```

An update is just a delete followed by an insert, assuming the deletion was successful.

A symmetric table implementation should also address systems issues such as how locking and logging are performed on rows of such tables. These issues depend on the locking and logging protocols used, and are beyond the scope of this paper.

[4] In general, X and Y are *vectors* (of equal length) of type-compatible attributes. For clarity of presentation, we shall omit vector notation, and employ examples in which X and Y are single attributes.

The formalism above allows multiple disjoint pairs of symmetric attributes. Thus, if R is symmetric in X, Y and also symmetric in V, W, it makes sense to talk about $ker_{XY}(R)$, $ker_{VW}(R)$, and $ker_{XY}(ker_{VW}(R)) = ker_{VW}(ker_{XY}(R))$. We can also generalize symmetry to more than two attributes.

Definition 2.2 *A relation $R(Z_1, \ldots, Z_n)$ is symmetric in $Z_1, \ldots, Z_k$ when $R(Z_1, \ldots, Z_n)$ holds if and only if for every permutation P of $Z_1, \ldots, Z_k$, $R(P(Z_1), \ldots, P(Z_k), Z_{k+1}, \ldots, Z_n)$ holds. Each such $R(P(Z_1), \ldots, P(Z_k), Z_{k+1}, \ldots, Z_n)$ is a* symmetric complement *of $R(Z_1, \ldots, Z_n)$. We define $ker_{Z_1,\ldots,Z_k}(R)$ to include only those tuples from R with $Z_1 \leq \ldots \leq Z_k$.* □

Indexing

Indexing of all symmetric attributes in $ker(R)$ should be done in a single index structure, so that a single index lookup suffices to find tuples with some symmetric attribute equal to a given probe value.

2.1 Normalization

Database normalization and the proposed kernel representation both aim to remove redundancy. However, normalization may be hampered by the presence of symmetry in the data.

Example 2.1 *Consider a database describing meetings of pairs of people that take place in certain locations at certain times, as in Example 1.1. Suppose that the initial database design has the schema $U(P_1, P_2, L, D, T, A)$, where P_1 and P_2 are the parties, L is the location, D is the date, and T is the time. A is a law-enforcement agent assigned to monitor the meeting, and multiple agents can be assigned to a single meeting. The schema designer is aware that the database system provides facilities for symmetric relations, and wishes to take advantage of these facilities by declaring U to be symmetric in P_1, P_2.*

Suppose that there can be only one meeting that takes place in a given location on a given date and time. The symmetric redundancy prevents the expression of functional dependencies having LDT on the left hand side. As a result, the "obvious" normalization of the table into the meets *relation $M(P_1, P_2, L, D, T)$ and the* monitors *relation $S(L, D, T, A)$ is missed.* □

The solution to the problem identified in Example 2.1 is to apply the kernel first, and then try to normalize the result using standard normalization techniques. In Example 2.1, it is possible to identify the functional dependency $LDT \rightarrow P_1P_2$ in $ker_{P_1P_2}(U)$. This functional dependency allows the normalization of $ker_{P_1P_2}(U)$ into $ker_{P_1P_2}(M)$ and S; M is represented as a symmetric relation.

2.2 Implementation

It is straightforward to implement the γ operator. For each input tuple output that tuple in addition to tuples formed by permuting the symmetric attributes (but don't output a tuple twice if two permutations generate the same tuple). However, in a practical database system, the mapping from algebraic operators to implementations is not necessarily a direct one. For example, it is common to implement a scan operator with predicates, so that the `getnext` function returns the next row satisfying the predicates. This choice allows the scan operator to choose an appropriate access structure, such as an index if one exists.

In a similar way, the natural implementation of symmetric closure should also incorporate predicates on the symmetric attributes. The predicates allow for the efficient use of available access methods, and may avoid the generation of permutations that will be immediately filtered out. The predicates may come from selection operators or from join operators.

Example 2.2 *Consider again the M table from Example 2.1, in which the P_i attributes store the identifiers of persons involved in a pairwise meeting. Suppose that $\sigma_{P_2=456}(M)$ is a subexpression of a query to be evaluated. Let $K = ker(M)$ be stored by the database, so that the subexpression can be evaluated as $\sigma_{P_2=456}(\gamma_{P_1P_2}(K))$. Suppose also that we store a single index structure for the columns P_1 and P_2. For simplicity of presentation, assume that the database knows that for all rows of M, $P_1 \neq P_2$.*

Then by implementing an operator for the combined selection and symmetric closure, we can directly look up tuples in K having 456 for either of the symmetric attributes, and for each match return the permutation with $P_2 = 456$. The alternative permutation is never generated.

If we implemented symmetric closure as a stand-alone operator, then the best we could do would be to rewrite $\sigma_{P_2=456}(\gamma_{P_1P_2}(K))$ as $\sigma_{P_2=456}(\gamma_{P_1P_2}(\sigma_{P_1=456 \vee P_2=456}(K)))$. (See Lemma 3.2 below.) The pushed selection conditions allow the use of the index on K. However, both permutations of each matching row in K are generated, one of which will be filtered by the outer selection condition. □

In the general case for Example 2.2, it is possible that $P_1 = P_2$. A limited form of duplicate elimination would then be needed to avoid generating an output row twice from a single input row. Also observe that the problems highlighted by Example 2.2 become worse for symmetric relations over more than two attributes.

For a fixed number of symmetric columns, the symmetric closure operator can be expressed in relational algebra in terms of the union and attribute-renaming operators. Thus neither γ nor the kernel operator add

to the expressive power of relational algebra. Nevertheless, by abstracting the γ operator one can derive implementations directly for γ (or γ together with selection). The situation is analogous to the join operation which, though expressible in terms of selection and cartesian product, is best implemented directly.

3 Query Optimization

Given a query that mentions a symmetric relation R, we assume that we have physically stored just $K = ker(R)$. In an algebraic expression for a query that accesses R, we use $\gamma(K)$ in place of R.

In order to minimize the size of intermediate results, it would be beneficial to push other operators inside the symmetric closure operator γ, where possible. To support such an endeavor, we now describe algebraic equivalences that can form the basis of such rewriting rules. For simplicity of presentation, we phrase these rules for binary symmetric relations. Generalizations to higher symmetric arity are possible.

For the following results, we assume that S_1 and S_2 are arbitrary relations with attributes including X and Y, such that all rows satisfy $X \leq Y$. T represents an arbitrary relation that does not have attributes X or Y. Except for Lemma 3.6, the equivalences hold under both a set semantics and a multiset semantics (in which duplicate rows are permitted) for relations.

Definition 3.1 *Let θ be a condition on X and Y, and (possibly) other attributes. Let θ' be formed from θ by substituting X for Y and vice versa. We say that θ is a* symmetric condition *on X and Y if $\theta \equiv \theta'$. Given a nonsymmetric condition θ, we call the condition $\theta \vee \theta'$ the* symmetric closure *of θ, which we denote by $\hat{\theta}$ when the attributes X and Y are clear from context.* □

Example 3.1 *Symmetric selection conditions on X and Y include $X = Y$, $X^2 + Y^2 = 1$, and any condition that mentions neither X nor Y. Symmetric join conditions on $R.X$ and $R.Y$ include $R.X = S.A \wedge R.Y = S.A$, $R.X^2 + R.Y^2 = S.A^2$, and conditions that do not mention $R.X$ or $R.Y$. The condition $R.X - R.Y > 7$ is not symmetric; its symmetric closure is $R.X - R.Y > 7 \vee R.Y - R.X > 7$.* □

Symmetric conditions can be pushed below the symmetric closure.

Lemma 3.1 *If θ is a symmetric condition, then*

- $\sigma_\theta(\gamma_{XY}(S_1)) = \gamma_{XY}(\sigma_\theta(S_1))$
- $\gamma_{XY}(S_1) \bowtie_\theta T = \gamma_{XY}(S_1 \bowtie_\theta T)$

□

Because the symmetric closure of a condition is always symmetric, Lemma 3.1 implies the following result, which allows us to push down partial information from selections on the symmetric attributes.

Lemma 3.2 *For an arbitrary condition θ,*

- $\sigma_\theta(\gamma_{XY}(S_1)) = \sigma_\theta(\gamma_{XY}(\sigma_{\hat{\theta}}(S_1)))$
- $\gamma_{XY}(S_1) \bowtie_\theta T = \sigma_\theta(\gamma_{XY}(S_1 \bowtie_{\hat{\theta}} T))$

□

Lemma 3.3 *Suppose θ is a condition that implies $X \leq Y$. Then $\sigma_\theta(\gamma_{XY}(S_1)) = \sigma_\theta(S_1)$.* □

Lemma 3.4

- $\gamma_{XY}(S_1) \cup \gamma_{XY}(S_2) = \gamma_{XY}(S_1 \cup S_2)$
- $\gamma_{XY}(S_1) \cap \gamma_{XY}(S_2) = \gamma_{XY}(S_1 \cap S_2)$
- $\gamma_{XY}(S_1) - \gamma_{XY}(S_2) = \gamma_{XY}(S_1 - S_2)$
- $\gamma_{XY}(S_1) \times T = \gamma_{XY}(S_1 \times T)$

□

Lemma 3.5 *If attribute list G includes both X and Y, then $\pi_G(\gamma_{XY}(S_1)) = \gamma_{XY}(\pi_G(S_1))$.* □

Lemma 3.6 *Under a set semantics: (a) If attribute list G includes neither X nor Y, then $\pi_G(\gamma_{XY}(S_1)) = \pi_G(S_1)$. (b) If attribute list G includes X but not Y, and if G' is the same as G except that X is replaced by Y, then $\pi_G(\gamma_{XY}(S_1)) = \pi_G(S_1) \cup \pi_{G'}(S_1)$.* □

Definition 3.2 *Let $\mathcal{A}^{\vec{f}}_G(R)$ denote the aggregate of relation R, grouped by the columns in the list G, computing the aggregate functions $\vec{f}$.* □

Lemma 3.7 *If grouping attributes G include both X and Y, then $\mathcal{A}^{\vec{f}}_G(\gamma_{XY}(S_1)) = \gamma_{XY}(\mathcal{A}^{\vec{f}}_G(S_1))$* □

Aggregates grouping by X alone or Y alone can use Lemma 3.7 to first compute the aggregate grouped by X and Y. Assuming that the aggregate functions are incrementally computable, the coarser aggregates can then be computed in a subsequent operation.

Lemma 3.8 *Let G be grouping attributes other than X and Y, and let $\vec{f}$ contain just idempotent aggregates such as* `min` *and* `max`*. Then $\mathcal{A}^{\vec{f}}_G(\gamma_{XY}(S_1)) = \mathcal{A}^{\vec{f}}_G(S_1)$.* □

It is tempting to think of analogous equivalences to those of Lemma 3.8 for other aggregates. However, a row in the kernel maps to either one or two rows in the symmetric closure, depending on whether the symmetric attributes have equal values. To take account of this difference, one can split the kernel into two fragments.

Lemma 3.9 *Let G be grouping attributes other than X and Y, and let $\vec{f}$ contain just linear aggregates such as* `sum` *and* `count`*. Let $2\vec{f}$ denote the aggregate that computes double the aggregate functions $\vec{f}$. Then*

- $\mathcal{A}_G^{\vec{f}}(\gamma_{XY}(\sigma_{X<Y}(S_1))) = \mathcal{A}_G^{2\vec{f}}(\sigma_{X<Y}(S_1))$
- $\mathcal{A}_G^{\vec{f}}(\gamma_{XY}(\sigma_{X=Y}(S_1))) = \mathcal{A}_G^{\vec{f}}(\sigma_{X=Y}(S_1))$

□

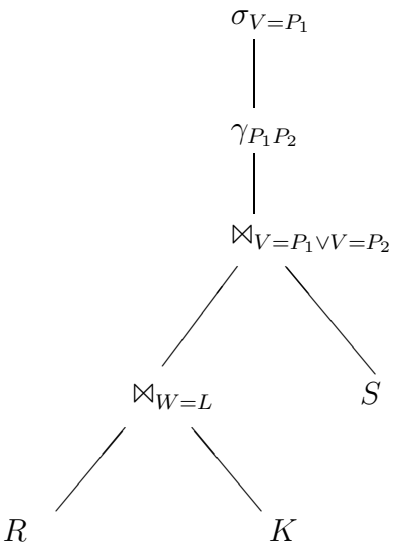
For incremental aggregate functions, one can compute $\mathcal{A}_G^{\vec{f}}(\gamma(S_1))$ by partitioning S_1 into two pieces, using Lemma 3.9.

To be able to compose the various lemmas above, we need to verify that in each case the subexpression created by pulling γ up one level retains the property that $X \leq Y$. This verification is straightforward, and is omitted here.

Example 3.2 *Consider again the "meets" relation $M(P_1, P_2, L, D, T)$ from the law-enforcement database of Example 2.1. Let $S(V, \ldots)$ be another relation indicating that person V is a suspect. Let $R(W, \ldots)$ be a relation indicating that W is a location being monitored. The law-enforcement user poses the query*

$$S \bowtie_{V=P_1} M \bowtie_{W=L} R$$

to find meetings involving suspects at monitored locations. Suppose M is stored in the database as $K = ker_{P_1P_2}(M)$. Using Lemmas 3.1 and 3.2, the query can be rewritten so that it has the following tree form:



The joins are pushed below the γ operator, one fully and the other partially. When the joins are selective, the rewritten plan is more efficient than one in which γ is applied directly to K, because (a) the symmetric closure is applied to fewer tuples, and (b) the operators below γ are applied to fewer tuples. □

Example 3.3 *Suppose that in Example 2.1 we simply wish to list all records in the meets relation, without redundancy. The user writes the query as*

$$\sigma_{P_1 \leq P_2} M.$$

If $K = ker_{P_1P_2}(M)$, then the query can be rewritten as $\sigma_{P_1 \leq P_2}(\gamma_{P_1P_2}K)$. By Lemma 3.3, this expression is equivalent to $\sigma_{P_1 \leq P_2}K$. One can even use semantic query optimization to observe that $P_1 \leq P_2$ is an integrity constraint on K, which allows further simplification of the query to simply K. Thus, even if the user does not query the kernel, appropriately formulated queries over the symmetric closure can achieve the performance that would have been available by querying the kernel directly. □

3.1 Conditions Revisited

Consider again the "meets" relation M, and suppose that we wish to identify pairs of meetings that share one (or more) members. Assume there is no index available on M. Using subscripts to distinguish two instances of M, we might write this query as

$$M_1 \bowtie_{M_1.P_1 = M_2.P_1} M_2 \quad (Q1)$$

This query cannot be effectively optimized, because all permutations of tuples in the two instances of M need to be generated to test the condition on P_1.

Now imagine we defined a function $overlap_{R,S}$, where R and S are symmetric relations on $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$ respectively. Suppose that r and s represent rows of R and S respectively. Applied to (r, s), $overlap_{R,S}$ returns the cardinality of the multiset intersection of $\{r.X_1, \ldots, r.X_n\}$ and $\{s.Y_1, \ldots, s.Y_m\}$. In other words, $overlap_{R,S}$ returns the number of common values among the symmetric attributes of the two rows. The *overlap* function can be implemented efficiently, even for symmetric relations of high arity, by sorting the symmetric attributes from each tuple and then scanning through each list.

Given this *overlap* function, we can rephrase the query above as

$$\sigma_\theta(M_1 \bowtie_{overlap_{M_1,M_2} \geq 1} M_2) \quad (Q2)$$

Here θ is the condition $M_1.P_1 \leq M_1.P_2 \wedge M_2.P_1 \leq M_2.P_2$; like in Example 3.3, the outer selection removes redundant copies of qualifying rows. This query *can* be optimized, because the join condition is symmetric. After several transformations, the query becomes

$$K_1 \bowtie_{overlap_{K_1,K_2} \geq 1} K_2$$

where $K = ker_{P_1,P_2}(M)$. The *overlap* function can be implemented particularly efficiently on kernels, because the symmetric attributes are already in order.

There is a difference between formulations Q1 and Q2 of the query: Q1 requires the first symmetric attribute of each component row to be the common member, while Q2 does not.[5] Thus, if the user doesn't

[5] Also, Q1 will output two rows for pairs of meetings between the same two people, while Q2 outputs one.

need a special way of identifying the common member, then it pays (in terms of query execution time) to use the formulation Q2. The trick is to formulate the query using properties of the *set* of symmetric attributes where possible, because such conditions are always symmetric and can be better optimized.

Finally, note that the $overlap_{R,S} \geq 1$ test can be expressed alternatively as $r.X_1 = s.Y_1 \vee r.X_1 = s.Y_2 \vee r.X_2 = s.Y_1 \vee r.X_2 = s.Y_2$. Nevertheless, we advocate the use of a specialized *overlap* function, because (a) it simplifies the job of identifying symmetric conditions for the query compiler, (b) for symmetric relations of higher arity the equivalent logical expressions become unwieldy, and (c) the *overlap* function can be used to test other kinds of set-oriented relationships, such as disjointness and subset relationships.

The arguments in favor of a special *overlap* function for join conditions extend also to selection conditions. Definition 3.1 (the symmetric closure of a condition) can be extended to k-ary conditions by taking the disjunction of all expressions formed by permuting the symmetric columns. Thus, in a symmetric relation of arity k with symmetric columns $P_1, \ldots, P_k$, the closure of $P_1 = 123$ is $P_1 = 123 \vee \cdots \vee P_k = 123$. The closure of $P_1 = 123 \wedge P_2 = 456$ has $k(k-1)$ disjuncts. These expressions are unwieldy, and are likely to hide optimization alternatives from a realistic query optimizer. Instead, we propose to represent the symmetric closure of $P_1 = 123$ as "$\{123\}$ *Among* $\{P_1, \ldots, P_k\}$" and the closure of $P_1 = 123 \wedge P_2 = 456$ as "$\{123, 456\}$ *Among* $\{P_1, \ldots, P_k\}$". This representation is compact, and can represent the closure of common conditions that equate an attribute with a constant. It also allows for easier recognition of efficient plans, such as using a common index on $\{P_1, \ldots, P_k\}$, by the query compiler.

4 Inferring Symmetry

Being able to infer that a subexpression is symmetric enables additional options for query optimization. Also, if we can infer that a materialized view is guaranteed to be symmetric, then we can choose to store it in the more compact form, saving space and query processing time.

To formulate the inference problem, we use the notion of a conjunctive query [9] to represent a view. An *ordinary* subgoal employs a table predicate, while a *built-in* subgoal employs an interpreted predicate, such as equality or "<". An ordinary subgoal is *symmetric* if its predicate is a table that is marked as symmetric in the database schema. For ease of presentation we shall assume that we are dealing with binary symmetric relations whose symmetric attributes are the leftmost attributes as written.

Definition 4.1 *Let*

$$Q(X, Y, \vec{Z}) : -B(X, Y, \vec{Z}, \vec{W})$$

be a conjunctive query, where B is a conjunction of subgoals. $Q^T(X, Y, \vec{Z})$ is the conjunctive query defined by

$$Q^T(X, Y, \vec{Z}) : -B(Y, X, \vec{Z}, \vec{W})$$

with X and Y interchanged in B. □

Note that $(Q^T)^T = Q$, and that containment mappings from Q to Q^T are isomorphic to containment mappings from Q^T to Q.

Lemma 4.1 *Let Q be a conjunctive query containing nonsymmetric ordinary subgoals, and no built-in subgoals. Q is symmetric if and only if there exists a containment mapping from Q to Q^T.* □

Example 4.1 *Let $E(K, M)$ represent an employee relation, where K is the unique key of the employee, and M is the employee's manager. Let Q be the conjunctive query*

$$Q(K_1, K_2) : -E(K_1, M), E(K_2, M).$$

Q^T is then

$$Q^T(K_1, K_2) : -E(K_2, M), E(K_1, M)$$

and the identity mapping is a containment mapping from Q to Q^T. We therefore conclude that Q is symmetric in K_1 and K_2. □

When subgoals may themselves be symmetric, a simple containment mapping is not sufficient, as illustrated by the following example.

Example 4.2 *Consider again table E from Example 4.1. Let S be a symmetric relation; think of $S(M_1, M_2)$ as meaning that M_1 and M_2 are siblings. Let Q be the conjunctive query*

$$Q(K_1, K_2) : -E(K_1, M_1), S(M_1, M_2), E(K_2, M_2).$$

Q is indeed symmetric. However, Q^T is

$$Q^T(K_1, K_2) : -E(K_2, M_1), S(M_1, M_2), E(K_1, M_2)$$

and the identity mapping is not a containment mapping from Q to Q^T. The mapping that interchanges M_1 and M_2 is not a containment mapping, because $S(M_1, M_2)$ in Q maps to $S(M_2, M_1)$ in Q^T. □

Definition 4.2 *Let h be a symbol mapping from a conjunctive query $Q : -B$ to a conjunctive query $Q' : -B'$. We say that h is a* symmetric containment mapping *from Q to Q' if $h(Q) = Q'$, and for every subgoal S in B, either (a) $h(S)$ appears in B', or (b) S is symmetric and the symmetric complement of $h(S)$ appears in B', or (c) S is a built-in subgoal, and $h(S)$ is equivalent to a subgoal of B'.* □

Unlike parts (a) and (b), part (c) of Definition 4.2 is not syntactic identity; it depends on the proof system available to demonstrate equivalence. Part (c) allows us to identify symmetric conditions (Definition 3.1) in a logic-based formalism.

Lemma 4.2 *Let Q and Q' be conjunctive queries with ordinary subgoals that may be symmetric, and no built-in subgoals. Q is contained in Q' if and only if there exists a symmetric containment mapping from Q' to Q.* □

Lemma 4.3 *Let Q be a conjunctive query containing ordinary subgoals that may be symmetric, and no built-in subgoals. Q is symmetric if and only if there exists a symmetric containment mapping from Q to Q^T.* □

Lemma 4.3 resolves the difficulty of Example 4.2, because the mapping that interchanges M_1 and M_2 is a symmetric containment mapping. As in the nonsymmetric case [9], when built-in subgoals are allowed we lose the "only-if" part of Lemma 4.3.

Lemma 4.4 *Let Q be a conjunctive query containing ordinary subgoals that may be symmetric, and built-in subgoals. Q is symmetric if there exists a symmetric containment mapping from Q to Q^T.* □

4.1 Optimization using Inference

Suppose we can infer that a query subexpression is guaranteed to be symmetric. Then we can deliberately insert a "kernelization" operation paired with a symmetric closure operation, and move the predicates around to minimize the size of intermediate results. Thus we can benefit from the proposed query optimization techniques of Section 3 even if we do not have any stored kernels in the database.

Example 4.3 *Consider again table E from Example 4.1. We write E_1 and E_2 to distinguish two instances of E in a single query, and we similarly subscript the attributes of E. Consider a query*

$$(E_1 \bowtie_{M_1=M_2} E_2) \bowtie_{\theta_1} R_1 \ldots \bowtie_{\theta_m} R_m.$$

Suppose that none of the θ_i conditions mention K_1 or K_2. We begin by inferring that the subexpression $(E_1 \bowtie_{M_1=M_2} E_2)$ is symmetric in K_1 and K_2; see Example 4.1. We can therefore rewrite the query as

$$\gamma_{K_1,K_2}(\sigma_{K_1 \le K_2}(E_1 \bowtie_{M_1=M_2} E_2)) \bowtie_{\theta_1} R_1 \ldots \bowtie_{\theta_m} R_m.$$

By repeatedly applying Lemma 3.1, this is equivalent to

$$\gamma_{K_1,K_2}(\sigma_{K_1 \le K_2}(E_1 \bowtie_{M_1=M_2} E_2) \bowtie_{\theta_1} R_1 \ldots \bowtie_{\theta_m} R_m)$$

which is more efficient than the original expression because the intermediate joins are smaller. □

5 Extending SQL

In this section, we extend SQL with features that allow the expression of bounded-cardinality multisets as database columns. Our extended SQL can be translated into the algebra described previously. The proposed syntactic constructs enable the succinct expression of queries that manipulate bounded multisets. Further, specialized syntax for commonly used operations can help the database system choose efficient query processing algorithms to execute the query [2, 11].

When creating a table, one may declare k columns of the same type to be a named *multiset*. This declaration serves two purposes. It provides a name for the group of attributes that can be used in writing queries. It also gives a hint to the database system to create an index on the union of all columns in the group. The multiset may optionally be declared to be symmetric, in which case the database system is free to permute the columns (e.g., to store the kernel) to make integrity constraint checking and query processing more efficient.

Example 5.1 *In Example 1.4, a multiset* `Persons` *would be declared for the columns containing the (integer) identifiers of persons participating in the meeting, and* `Persons` *would be declared symmetric.*

```
Create Table M (
  Meeting-id integer,
  Symmetric Multiset Persons
    { P1, ..., Pk } integer,
  ... )
```

In Example 1.5, a multiset `Slots` *would be declared for the columns C_1 through C_{24}.* `Slots` *would not be declared symmetric.*

```
Create Table V (
  ID integer,
  ViewDate date,
  Multiset Slots
    { C1, ..., C24 } integer,
  ... )
```

In these examples, users may query the attributes `Pi` *and* `Ci` *directly as regular attributes, using standard SQL syntax.* □

We introduce new "column variables" that are allowed to take values from any one of a set of columns. The original columns of a table are not permuted. This choice allows us to access a symmetric base table T directly in the conventional way, without forcing the query "Select * from T" to have $k!$ copies of each tuple representing a k-element multiset. The scope of a column variable is defined using the `Among` keyword in the `Where` clause.[6]

[6]The occurrences of column variables must be *safe* in the sense of [10].

Example 5.2 *Consider Example 1.5 together with the sample query "Find all individuals who, on the given date, have watched channels 2 and 4, but not channel 5." We would write this query as*

```
Select ID, ViewDate
From V
Where {X1,X2} Among Slots and X1=2 and X2=4
  and not ({5} Among Slots)
```

There is one row per `ID` *and* `ViewDate` *in the output, even though there may be many possible combinations of slots satisfying the conditions in the* `Where` *clause.* □

When we write `{X1,X2} Among Slots` it is implicit that `X1` and `X2` correspond to different columns within `Slots`. If `X` is a column variable, we use the syntax `X.name` to denote the column name of the column actually bound to `X` in the query. One can use the `Among` keyword for groups not explicitly defined as multisets by explicitly listing the columns, as in "`{X1,X2} Among {Jan,Feb,Mar,Apr}`".

Example 5.3 *Continuing Example 5.2, suppose that we include a column variable in the* `Select` *clause.*

```
Select ID, ViewDate, X1.name, X1
From V
Where {X1,X2} Among Slots and X1=2 and X2=4
  and not ({5} Among Slots)
```

Unlike before, there are multiple rows per `ID`/`ViewDate` *in the output, one for each binding of* `X1` *to a column whose value (together with some* `X2` *value) satisfies the conditions of the* `Where` *clause. The column-variables in the select clause implicitly control duplicate elimination. Since only* `X1` *is mentioned in the select clause, there is one value output for each* `X1` *column binding, irrespective of how many valid* `X2` *values are present.* □

Example 5.3 shows how to "unpivot" a k-element multiset from a column-based representation into a more traditional row-based representation. One could use variants of Example 5.3 to define views over which traditional SQL methods of set manipulation can be expressed. As a result, none of SQL's expressive power for set manipulation has been lost by using a column-wise representation. We emphasize that since the unpivoted table is just a view, queries over the unpivoted table could be translated into queries over the original (pivoted) table, which may be more efficient because joins are not required.

Example 5.4 *Consider Example 1.4 in which we have a meeting table M with k attributes $P_1, \ldots, P_k$ grouped into a multiset called* `Persons`*. We wish to find all pairs of people X and Y at three degrees of separation. In other words, we need three meetings M_1, M_2, M_3 such that X attended M_1, Y attended M_3, M_1 and M_2 have overlapping membership, and M_2 and M_3 have overlapping membership. We can write this query as*

```
Select X, Y
From M M1, M M2, M M3
Where {X} Among M1.Persons
  and {W} Among M1.Persons
  and {W} Among M2.Persons
  and {Z} Among M2.Persons
  and {Z} Among M3.Persons
  and {Y} Among M3.Persons
```

`{W} Among M2.Persons` *and* `{Z} Among M2.Persons` *are written separately, meaning that* `W` *and* `Z` *may bind to the same column. Had we written* `{W,Z} Among M2.Persons`*, they would have to be different columns. One could also formulate the query succinctly using an "overlap" method, as discussed in Section 3.1:*

```
Select X, Y
From M M1, M M2, M M3
Where {X} Among M1.Persons
  and Overlap(M1.Persons,M2.Persons) >= 1
  and Overlap(M2.Persons,M3.Persons) >= 1
  and {Y} Among M3.Persons
```

□

Without the `Among` syntax, there would be no way to output values from multiple columns in a single select statement. One would need to form the union of k^2 select statements to express Example 5.4.

A conventional set representation would require a six-way join to express Example 5.4.

When a symmetric multiset has fewer elements than the cardinality bound, the remaining columns are padded with NULLs. Column-variables cannot be bound to NULL values.

We also advocate additional syntactic elements for directly expressing multisets formed as the intersection or difference of other multisets. (Note that union of two k-bounded multisets is not necessarily a k-bounded multiset.)

The translation of the extended SQL into the extended algebra is relatively straightforward. When symmetric attributes are referenced using the `Among` keyword, the underlying relation has its symmetric columns copied into new columns. Some of these new columns correspond to the column variables. The symmetric closure operator is applied to the new columns to find combinations of values satisfying the conditions on column variables in the `Where` clause. An algebraic duplicate-elimination step is also needed, as is special handling for NULL values. After the query has been translated, it can be optimized and executed as outlined in Sections 2.2 and 3.

6 Experimental Evaluation

In this section we describe an experimental evaluation of various representations of multisets on a state-of-the-art commercial database system. We wish to demonstrate the qualitative performance characteristics of various representations. A comprehensive performance evaluation is beyond the scope of this paper.

We consider a database of randomly generated 3-element multisets, where each element is a string chosen uniformly from a set of about 8,000 English words. The schema of the kernel table K is (X_1, X_2, X_3, Y) where X_1, X_2, X_3 are the set elements. We construct K so that $X_1 \leq X_2 \leq X_3$, and create an index on X_1, an index on X_2, and an index on X_3. We store 500,000 such sets in the database.

We define a view V over K as the union of all six permutations (each expressed using a select statement) of X_1, X_2, X_3 from K.

We also store a conventional set-based representation of the same data in which a new set-identifier attribute ID is defined. We create one table $S(ID, Y)$, and another $M(ID, X)$ containing the unpivoted sets. An index on X in M is created.

We consider four variants of a query that finds sets with all three members specified by constants. In the first variant Q_1, we query K for some combination of attributes.[7]

```
Select X1,X2,X3,Y
From K
Where (X1='foo' and X2='bar' and X3='baz')
   or (X1='foo' and X3='bar' and X2='baz')
   or (X2='foo' and X1='bar' and X3='baz')
   or (X2='foo' and X3='bar' and X1='baz')
   or (X3='foo' and X1='bar' and X2='baz')
   or (X3='foo' and X2='bar' and X1='baz')
```

In the second variant Q_2 of the query, we write the query in terms of V.

```
Select X1,X2,X3,Y
From V
Where (X1='foo' and X2='bar' and X3='baz')
```

In the third variant Q_3, we query the conventional set-based representation.

```
Select M1.X, M2.X, M3.X, S.Y
From S, M M1, M M2, M M3
Where (M1.X='foo' and M2.X='bar'
       and M3.X='baz') and
 M1.ID=S.ID and M2.ID=S.ID and M3.ID=S.ID
```

Our extended syntax for the query would be

```
Select X1,X2,X3,Y
From K
Where {'foo','bar','baz'} Among {X1,X2,X3}
```

Our proposed access plan (use a combined index on all set columns) is not directly supported by the database system. Thus, the best we can do is to construct a query Q_4 whose performance is likely to be comparable to our intended query plan. (We need to verify that the chosen plan for Q_4 is similar to our intended plan.) Q_4 is

```
Select X1,X2,X3,Y
From K
Where (X1='bar' and X2='baz' and X3='foo')
```

in which the constants are selected in alphabetical order.

We ran each of these queries using a commercial database system on a 1.4GHz Intel Centrino machine under Windows XP. We record the optimization time and execution time as reported by the database system. These are elapsed-time measurements. Each query was run on a cold database that had just been started. The numbers below reflect the average of five runs for each query. In each run a different combination of constants was used, and the combinations were chosen so that there was always a match in the database. The database system also reported the plan chosen to execute each query.

The plan chosen for Q_4 uses the indexes on `X1`, `X2` and `X3` to find matching row identifiers, intersects the set of identifiers, and finds rows from K for the matching identifiers. Our intended plan would do the same operations, but using a single common index for `X1`, `X2` and `X3`. While the number of row identifiers being intersected may be higher with our proposed method, the performance of Q_4 should roughly approximate the performance of our proposed method.

The plans chosen for Q_1 and Q_2 are similar to each other, consisting of the union of 6 subplans of the form mentioned for Q_4, one subplan per permutation of the attributes.

The plan chosen for Q_3 was a tree of three index-nested-loops joins. The innermost (i.e., leftmost) table is `M` accessed using an index lookup based on the `X` column. The other three index lookups are on the `ID` attributes of `M` (twice) and `S`.

Figure 1 shows the actual execution time for each of the four queries as reported by the database system. Figure 1 does not include the query optimization time, which is shown separately in Figure 2.

Figure 1 shows that the execution cost of Q_4 is smallest, with Q_1 and Q_2 having comparable execution cost. The cost of Q_3 is about 35 times higher than Q_4.

Figure 2 shows that the optimization cost of all three queries is comparable, although Q_2 has a noticeably lower optimization cost. This lower optimization

[7] In general we cannot take advantage of the order of constants mentioned in the query since the constants we're looking for may be bound at query time, and since we may be querying on just a subset of the available columns.

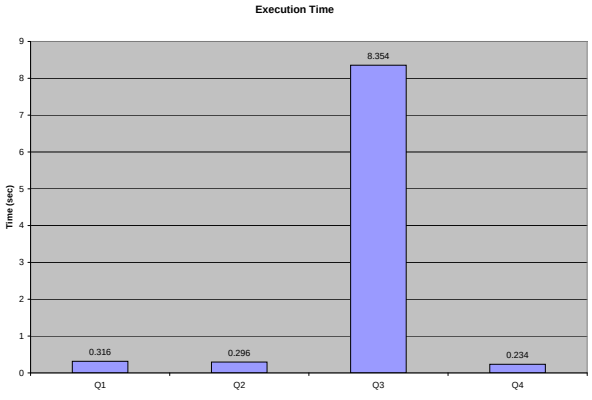


Figure 1: Execution time of the four queries.

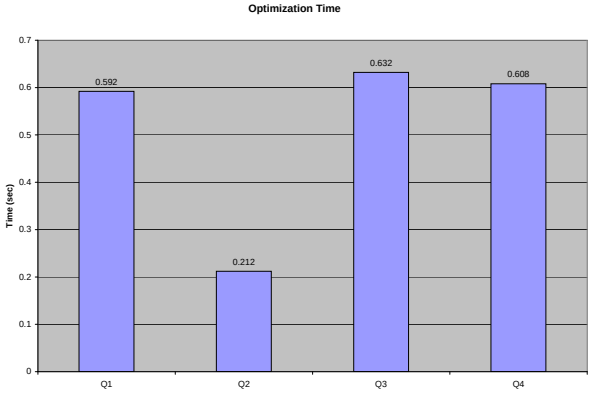


Figure 2: Optimization time of the four queries.

cost is probably just an artifact of a smaller search space of plans within the query optimizer, and not something intrinsic to the query itself. (Note the importance of separating the optimization time from the execution time in interpreting these results. Had we just reported the total elapsed time, Q_2 would have been the winner.)

Of the four solutions (Q_1, Q_2, Q_3, and our proposed method), only our method scales with the number of attributes. Q_3 does not scale because it requires a k-way join for multisets containing k elements. As one can see in Figure 1, even for $k = 3$ the performance of Q_3 is more than an order of magnitude worse than competing approaches.

Suppose that writing a basic condition (of the form `table.attribute=value`) takes 10 bytes of memory. If we try to generalize Q_1 and Q_2 to k-element multisets, then they require either a query or view definition whose size is approximately $10k(k!)$ bytes. The impact of this rate of growth is shown in Figure 3; note the logarithmic vertical scale. Q_1 and Q_2 quickly become impractical: with $k = 11$ the space for the query/view definition alone is four gigabytes.

In contrast, our query specification has size linear in k, and it can be evaluated without joins.

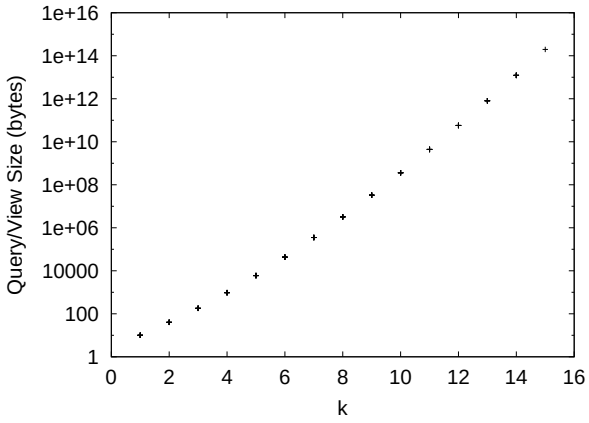


Figure 3: Growth of Q_1 and Q_2 with the multiset cardinality k.

7 Conclusions

We provide techniques that enable a database engine to support a symmetric table type. The techniques include

- A nonredundant data structure with update methods and specialized indexes.
- Methods for normalization in the presence of symmetric tables.
- An algebraic symmetric closure operator, together with algebraic equivalences useful for query optimization.
- Inference methods to determine when a query/view is guaranteed to be symmetric.
- A syntactic SQL extension to enable compact query expression.

A symmetric table type allows database schema designers to model symmetric relationships without having to worry about integrity, redundancy, consistency of updates, query efficiency, or suboptimal physical design.

One could go even further and implement different kinds of symmetric table. For example, the class of *antireflexive* symmetric relations (i.e., k-element sets rather than multisets) satisfies simpler algebraic rules, and some duplicate elimination steps can be omitted in the implementation of the γ operator (see Example 2.2).

We have argued that our approach is applicable when there is a natural cardinality bound in the application. One could extend our approach to general multisets by using a combined structure, i.e., a bounded cardinality multiset for an initial subset of elements, and a conventional set representation for additional elements.[8] An appropriate syntax could hide this di-

[8]This idea was suggested to us by Wisam Dakka.

vision from the user, presenting a single multiset abstraction. When small sets are typical, such an approach would have performance benefits, even in the absence of a strict cardinality bound.

References

[1] *Oracle Database Application Developer's Guide — Object-Relational Features*, 2004. 10g Release 1 (10.1), Part Number B10799-01.

[2] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the International Conference on Very Large Databases*, pages 295–306, 1996.

[3] C. J. Date. On various types of relations. *Database Debunkings*, April 20, 2003. Available at `http://www.dbdebunk.com/`.

[4] Jan Van den Bussche and Dirk Van Gucht. A hierarchy of faithful set creation in pure OODB's. In Joachim Biskup and Richard Hull, editors, *Database Theory - ICDT'92, 4th International Conference, Berlin, Germany, October 14-16, 1992, Proceedings*, volume 646 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 1992.

[5] Jan Van den Bussche and Dirk Van Gucht. The expressive power of cardinality-bounded set values in object-based data models. *Theoretical Computer Science*, 149(1):49–66, 1995.

[6] Sven Helmer and Guido Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *International Conference on Very Large Databases*, pages 386–395, 1997.

[7] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 157–168, 2003.

[8] Karthikeyan Ramasamy, Jignesh M Patel, Raghav Kaushik, and Jeffrey F Naughton. Set containment joins: The good, the bad and the ugly. In *International Conference on Very Large Databases*, pages 351–362, 2000.

[9] D. Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[10] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus. *ACM Trans. Database Syst.*, 16(2):235–278, 1991.

[11] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, and Sankar Subramanian. Spreadsheets in RDBMS for OLAP. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 52–63. ACM Press, 2003.