Using Horizontal-Vertical Decompositions to Improve Query Evaluation

Chris M. Giannella, Mehmet M. Dalkilic, Dennis P. Groth, Edward L. Robertson *

Indiana University Computer Science, Bloomington, IN 47405

{cgiannel, dalkilic, dgroth, edrbtsn}@cs.indiana.edu

Abstract

We investigate how relational restructuring may be used to improve query performance. Our approach parallels recent research extending semantic query optimization (SQO), which uses knowledge about the instance to achieve more efficient query processing. Our approach differs, however, in that the instance does not govern whether the optimization may be applied; rather, the instance governs whether the optimization yields more efficient query processing. It also differs in that it involves an explicit decomposition of the relation instance. We use approximate functional dependencies as the conceptual basis for this decomposition and develop query rewriting techniques to exploit it. We present experimental results using both synthetic and real-world data. These results lead to a characterization of a well-defined class of queries for which improved processing time is observed.

1 Introduction

A powerful feature of relational query languages is that identities of relational algebra may be used to transform query expressions to enhance efficiency of evaluation. These transformations are always valid, but whether they enhance or degrade efficiency depends upon

*All authors were supported by NSF Grant IIS-0082407

characteristics of the data. Furthermore, there are different transformations and different characteristics such that the validity of these transformations are dependent on these characteristics. This paper takes a characteristic of the second sort, namely functional dependency, recasts it to a characteristic of the first sort, and investigates the resulting implications on query evaluation.

Initially, functional dependencies were declared constraints, but dependencies that are discovered in particular instances may also be used in query optimization. The difficulty with using discovered functional dependencies is that they are brittle, so that a minor change to the instance may mean that a dependency no longer holds. Previous work (described below) has addressed this difficulty by trying to handle the situations in which a functional dependency may break. Our work, on the other hand, uses a more supple notion, approximate functional dependency (AFD), which bends but does not break as the instance changes. The notion of AFD applies to any instance, parameterized only by "degree of approximation." To our knowledge, this represents the first use of AFDs in query optimization. Indeed, approximate functional dependency is so ubiquitous that it disappears from discussion in most of the paper, replaced by a relational decomposition which, among other things, captures the "degree of approximation" in a simple manner.

This decomposition arises naturally from AFDs. The primary use of FDs is to decompose a table by projection, that is vertically. On the other hand, AFDs induce a partition into sets of tuples, that is horizontally, such that the FD holds in one of these partitions. These decompositions combine into a *Horizontal-Vertical decomposition* (HV decomposition), which is the heart of this paper, along with related query rewriting techniques that exploit the decomposition.

Query optimization as considered here involves modifying a query such that semantics is preserved but performance is enhanced. The first step is to replace the name of the decomposed relation by an expression that recovers the table from the decomposition. One would hope that off-the-shelf query evaluators could optimize the rewritten query, but unfortunately our experiments have failed to bear this out. Thus we have defined query rewrite rules that apply specifically to HV decomposed relations. As with other query rewriting, application of these rules may be blocked in a particular query, just as the standard "push down selects" rules is blocked when the select condition spans both branches of a join. We envision a scheme as shown in Figure 1; a query preprocessor that re-writes queries, taking advantage of the knowledge about the decomposition.

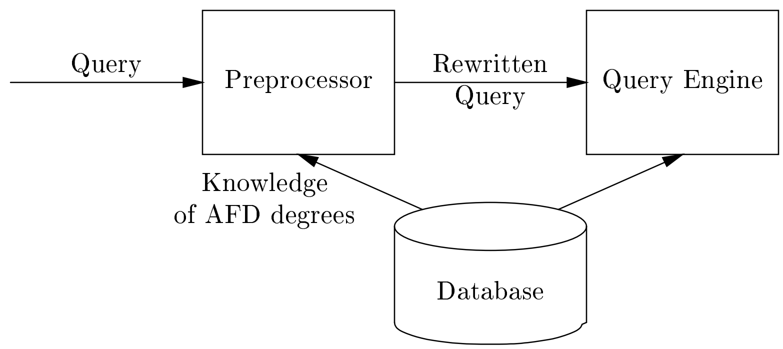

Figure 1: Queries are passed to a preprocessor that, using knowledge about the decomposition, affects the rewrite to make processing more efficient.

In order to appraise the decomposition and rewriting rules, we performed a number of experiments using synthetic and real-world data. Synthetic data allowed control of the "degree of approximation" of the AFD being tested. Real-world data was used to validate the synthetic data generation model. These experiments showed that rewriting yielded substantial performance improvements (on queries where they were not blocked, of course). Most surprisingly, these improvements occurred not only in cases where the AFD was quite close to an FD but also in cases that were far from an FD.

We envision our approach as only the beginning of a longer investigation of similar restructurings and rewritings. There are two different features that may vary in this investigation: classes of queries where rewritings may apply or be blocked and characteristics of instances that may suggest different restructurings. For example, multi-valued or approximate multi-valued decompositions are likely candidates.

The remainder of the paper is as follows. The next section provides a brief overview of the work leading up to our approach. Section 3 presents basic definitions and provides the theoretical background and results for the paper. Included in this are examples of two different query rewriting techniques. The results of experiments designed to test the effectiveness of the rewriting techniques are presented and discussed in Section 4. Section 5 describes an approach for maintaining HV decompositions. The paper ends with conclusions and future work.

2 Previous work: SQO and dependencies

Two separate trains of research led toward the work reported in this paper: a long-running and substantial effort in query optimization and a more recent interest in AFDs.

Semantic query optimization (SQO) began some two decades ago and was discovered independently by King[14, 15] and Hammer *et al.*[9]. SQO is the use of "semantic knowledge for optimizing queries..."[3, 12, 34]. Some researchers proposed that a query be rewritten prior to being passed to the query engine. The query is rewritten (to an equivalent query) according to a set of rewrite rules with the idea that the rewritten query will execute faster. One such rule allows sub-queries to be merged into the outer block (thereby eliminating the sub-query). This rewrite rule idea has been implemented in Starburst and IBM DB2 ([27, 28]).

Some researchers used additional information to rewrite queries. The most obvious candidates for use are integrity constraints (ICs)[35, 36], arbitrary predicates over the database that must hold for any instance (*e.g.* a primary key). In this context, ICs have been used to rewrite queries [24, 25, 30, 33, 34, 37]. For example, Paulley and Larson [24, 25] rewrite to eliminate unnecessary group by and distinct operations; Sun and Yu [37] rewrite to eliminate unnecessary joins, add beneficial restrictions, and eliminate redundant non-beneficial restrictions.

Functional dependencies (FDs) are a primary type of IC used in SQO. FDs possess a number of useful properties that can be exploited directly or indirectly. Although the use of FDs to induce lossless decompositions appears in basic database texts (*e.g.* [29]), there has been little work exploiting these decompositions for query optimization. Interestingly, as discussed by Cheng *et al.*[4], there does not appear to be any extensive commercial implementation of SQO in the spirit with which they were intended–beyond the typical keys and check constraints, likely because of their potentially complex nature, though convincing arguments exist for their use[31].

Fundamentally different from this use of declared constraints is that of discovering information about the *instance* itself that can be used in SQO. For example, several researchers have incorporated rules discovered from the data to rewrite queries [1, 10, 32, 39]. For example, Bell [1] uses discovered FDs to eliminate group by and distinct operations. Recently, work by Godfrey *et al.*[6, 7] demonstrates that instance knowledge yields significant, positive results in SQO. They use the concept of a soft constraint (SC), which reflects knowledge about the state of the database. Thus, SCs are weaker than traditional ICs in that they do not impose any conditions upon the database. The role they play, then, is not to ensure integrity, but to "semantically characterize the database"[7]. Godfrey *et al.* introduce two classes of SCs, absolute soft constraints (ASC) and statistical soft constraints[1] (SSC). ASCs hold completely and absolutely during the current state of the database. In contrast, SSCs do not hold completely. An obvious advantage of an ASC is that, when it holds, it can be incorporated in SQO, since, for the time it holds true, it functions essentially like an IC. ASCs can be applied to SQO for the purposes of: 1. query rewriting; 2. query plan parameterization; and 3. cardinality estimation. An advantage of an SSC is that it need not be checked against every update to verify whether it holds–rather, every so often the SSCs must be brought up to date. While SSCs can be useful for cardinality estimation, they cannot be used for query rewriting.

Godfrey *et al.* then describe how SCs would be incorporated into an RDBMS (since no current system is available): 1. discovery, 2. selection, 3. maintenance. For ASCs, Godfrey *et al.* focus on both checking when an ASC is violated and maintaining ASCs. Because of their tenuous nature *i.e.*, being state-dependent, considerable care must be given to both checking and maintaining ASCs – a difficult task. This is, in fact, the "Achilles heel" of ASCs. A natural question arises from the work of Godfrey *et al.*: how can SSCs be used, if at all, for query rewriting?

While the body of work deriving from SQO is substantial, a second, more recent body of work concerning AFDs was even more significant in the genesis of this paper.

The notion of a functional dependency was originally introduced as an IC for use in database design. However, more recently, research has been conducted with the view point

[1] The term "statistical" is meant to connote that the soft constraint holds true for some, if not all, of the data and not that any probabilistic techniques are used.

that FDs represent interesting patterns existent in the data. In this setting, FDs are not regarded as declared constraints. Researchers have investigated the problem of efficiently discovering FDs that hold in a given instance [11, 13, 16, 18, 20, 21, 23, 38]. Researchers have also considered the concept of an FD "approximately holding" in an instance and have developed measures to characterize the "degree of approximation". Piatetsky-Shapiro [26] describe a measure derived from probabilistic considerations (this measure corresponds to the τ measure of Goodman and Kruskal [8]). Kivinen and Mannila [16] propose and evaluate three different measures derived from pragmatic considerations. One of their measures, g_3, correlates with the idea of "correction" that we use. Huhtala *et al.* [11] develop an algorithm for efficiently discovering all AFDs in a given instance whose g_3 approximation measure is below a user specified threshold. Finally, approximation measures have been derived using information theoretic methods [2, 5, 17, 19, 22].

3 Definitions and Theoretical Results

In this section we describe the theoretical results that form the basis of our work. We start by defining our notation and describe the distinction we make between bags and sets. Then, we define the fundamental concept of a horizontal-vertical decomposition and describe some of its most important properties. After that, we describe two query rewriting techniques. The first technique is guaranteed to always preserves correctness. The second technique is guaranteed to preserve correctness only on a special class of queries described later. Finally, we close the section with two hypotheses about how our rewriting techniques affect query evaluation time.

We do not give proofs in this paper. Rather, we describe intuitively why the results work and illustrate with examples. Rigorous proofs can be given, but, in our opinion, do not illuminate the ideas.

A	B	C
1	2	1
1	3	1
2	1	1
2	1	1

Figure 2: An instance, $\mathbf{s}$, over schema $\{A, B, C\}$.

3.1 Basic Notation

Before describing our decomposition and query rewriting techniques, we introduce some basic notation. We assume the reader is familiar with the basic concepts of relational database theory (see [29] for a review). In what follows, we fix a relation symbol R with schema $\{A, B, C\}$. Our results can easily be generalized to apply to schema with any number of attributes, however, for simplicity, we stick with $\{A, B, C\}$.

Classical relational database theory is set-based (*e.g.* the relational algebra operators take sets and return sets). However, in practice, people work with tables rather than sets. One distinguishing property is that tables may contain repeats. This property impacts our work significantly and so cannot be ignored. Thus, when describing our theoretical foundations, we operate with bags rather than sets. Whenever we write "relation instance" or "instance" or "relation" we mean a bag and not necessarily a set. For example, the instance, $\mathbf{s}$, of R depicted in Figure 2 is a bag and not a set.

We also make the distinction between those relational algebra (RA) operators that return sets and those that return bags. The ones that return sets (*i.e.* the ones that are duplicate removing) are denoted in the standard way: Π, σ, $\bowtie$, δ, $\cup$, and $-$ (projection, selection, natural join, renaming, union, and minus, respectively). Their bag counterparts are denoted: $\hat{\Pi}$, $\hat{\sigma}$, $\hat{\bowtie}$, $\hat{\delta}$, $\hat{\cup}$, and $\hat{-}$. We call the relational algebra over all operators (bag and set) the *bag relational algebra (bag RA)*.

Let $\mathbf{s}_1, \mathbf{s}_2$ be instances over some schema S. We say that $\mathbf{s}_1$ and $\mathbf{s}_2$ are *set equivalent*, written $\mathbf{s}_1 \equiv \mathbf{s}_2$ if $\Pi_S(\mathbf{s}_1) = \Pi_S(\mathbf{s}_2)$. In other words, $\mathbf{s}_1$ and $\mathbf{s}_2$ are equal once duplicates have been removed. For example, consider the instance $\mathbf{s}$, depicted in Fig. 2; call the instance resulting from the deletion of the first tuple $\mathbf{s}_F$; call the instance resulting from the deletion of the last tuple (and not the first) $\mathbf{s}_L$. $\mathbf{s}$ is set equivalent to $\mathbf{s}_L$ but not $\mathbf{s}_F$.

Given, Q, a bag RA expression involving R, and E, another bag RA expression, let $Q[R \leftarrow E]$ be the result of replacing all occurrences of R by E. Note that the result may contain schema conflicts.

Example 1 *Let* $Q := \Pi_{A,B}(\sigma_{C=0}(R))$. *Let* $E := R_1 \hat{\cup} \hat{\Pi}_{A,B}(R_2)$ *where* R_1 *has schema* $\{A, B\}$ *and* R_2 *has schema* $\{A, B, C\}$. *Note that* E *has no schema conflicts. The schema of* E *is* $\{A, B\}$. *By definition* $Q[R \leftarrow E]$ *is* $\Pi_{A,B}(\sigma_{C=0}(E))$. *This expression has a schema conflict since* $\sigma_{C=0}$ *is being applied to* E *which has schema* $\{A, B\}$.

Consider another example. Let $Q' := \Pi_{A,B}(R \cup S)$ *(*S *has the same schema as* R, $\{A, B, C\}$*). By definition* $Q'[R \leftarrow E]$ *is* $\Pi_{A,B}(E \cup S)$. *This expression has a schema conflict since* E *has schema* $\{A, B\}$ *while* S *has schema* $\{A, B, C\}$.

$\square$

3.2 Horizontal-Vertical Decompositions

Let $\mathbf{r}$ be an instance of R. If the functional dependency $A \rightarrow B$ holds in $\mathbf{r}$,[2] then $\mathbf{r}$ may be decomposed vertically as $\mathbf{r}_{AB} = \Pi_{A,B}(\mathbf{r})$, $\mathbf{r}_{AC} = \hat{\Pi}_{A,C}(\mathbf{r})$. This decomposition enjoys the property of being join lossless: $\mathbf{r} = \mathbf{r}_{AB} \hat{\bowtie} \mathbf{r}_{AC}$.

If $A \rightarrow B$ does not hold in $\mathbf{r}$, then $\mathbf{r}$ may still be decomposed. But we must first horizontally decompose $\mathbf{r}$ into two disjoint, non-empty relation instances whose union is $\mathbf{r}$, such that $A \rightarrow B$ holds in one of these relation instances. This instance is then vertically decomposed. The result is a *horizontal-vertical decomposition* of $\mathbf{r}$. The next set of definitions makes this concept precise.

$\mathbf{r}^c \subseteq \mathbf{r}$ is said to be an $A \rightarrow B$ *correction* for $\mathbf{r}$ if $A \rightarrow B$ holds in $\mathbf{r} \hat{-} \mathbf{r}^c$. Often we omit mention of $A \rightarrow B$ and $\mathbf{r}$ when clear from context and only say that $\mathbf{r}^c$ is a correction. Given correction, $\mathbf{r}^c$, the horizontal-vertical decomposition (HV decomposition) induced is $\mathbf{r}'_{AB}$, $\mathbf{r}'_{AC}$, and $\mathbf{r}^c$ where $\mathbf{r}'$ is $\mathbf{r} \hat{-} \mathbf{r}^c$, $\mathbf{r}'_{AB}$ is $\Pi_{A,B}(\mathbf{r}')$, and $\mathbf{r}'_{AC}$ is $\hat{\Pi}_{A,C}(\mathbf{r}')$.

[2] for all tuples $t_1, t_2 \in \mathbf{r}$, $t_1[A] = t_2[A]$ implies $t_1[B] = t_2[B]$.

$\mathbf{s}^c$			$\mathbf{s}'_{AB}$		$\mathbf{s}'_{AC}$	
A	B	C	A	B	A	C
1	3	1	1	2	1	1
2	1	1	2	1	2	1

Figure 3: Induced HV non-minimal decomposition, $\mathbf{s}^c$, $\mathbf{s}'_{AB}$, $\mathbf{s}'_{AC}$.

$\mathbf{s}^c$			$\mathbf{s}'_{AB}$		$\mathbf{s}'_{AC}$	
A	B	C	A	B	A	C
1	3	1	1	2	1	1
			2	1	2	1
					2	1

Figure 4: Induced HV minimal decomposition, $\mathbf{s}^c$, $\mathbf{s}'_{AB}$, $\mathbf{s}'_{AC}$.

Recall the instance, $\mathbf{s}$, in Figure 2. Clearly, $A \to B$ does not hold. Let $\mathbf{s}^c = \{(1,3,1),(2,1,1)\}$. Since $A \to B$ holds in $\mathbf{s}' = \{(1,2,1),(2,1,1)\}$, then $\mathbf{s}^c$ is a correction. The HV decomposition induced is depicted in Figure 3. Notice that $\mathbf{s} = (\mathbf{s}'_{AB} \hat{\bowtie} \mathbf{s}'_{AC})\ \hat{\cup} \mathbf{s}^c$. Moreover, in this case, $\hat{\Pi}_{A,B}(\mathbf{s}) = \mathbf{s}'_{AB} \hat{\cup} \hat{\Pi}_{A,B}(\mathbf{s}^c)$. The first observation points to the lossless property of HV decompositions. The second observation points to another property that will later be shown important: if C is not needed in $\mathbf{s}$, then the join can be eliminated from the decomposition.

In the previous example, $\mathbf{s}^c$ is not minimal since a smaller correction can be found. For example, $\mathbf{s}^c = \{(1,3,1)\}$ is also a correction. The HV decomposition induced is depicted in Figure 4. Notice that $\mathbf{s} = (\mathbf{s}'_{AB} \hat{\bowtie} \mathbf{s}'_{AC})\ \hat{\cup} \mathbf{s}^c$. Moreover, $\hat{\Pi}_{A,B}(\mathbf{s}) \equiv \mathbf{s}'_{AB} \hat{\cup} \hat{\Pi}_{A,B}(\mathbf{s}^c)$ (but equality does not hold). As in the previous example, the first observation points to the lossless property of the decomposition, and the second observation points to the property that if C is not needed in $\mathbf{s}$, then the join can be eliminated from the decomposition. However, this example shows that the property must be weakened to set equality rather than equality (see Theorem 1).

The point of these last two examples was (i) to illustrate two important properties of HV decompositions, and (ii) to point out that these properties do not depend on the correction being minimal. Our query rewriting techniques rely on these properties. If these properties were dependent on the decomposition being minimal, then maintaining the decomposition in the presence of updates would be difficult. But, these properties are not dependent on the correction being minimal. Hence, we gain much greater maintenance flexibility. Nonetheless, maintenance is still a difficult issue. We describe a simple method for maintaining the decomposition in section 5, but, do not analyze its performance. We leave this as future work.

In short, the fundamental properties of HV decompositions needed for our query rewriting techniques are the following.

Theorem 1

1. $\mathbf{r} = (\mathbf{r'}_{AB} \hat{\bowtie} \mathbf{r'}_{AC}) \hat{\cup} \mathbf{r}^c$.
2. $\hat{\Pi}_{A,B}(\mathbf{r}) \equiv \mathbf{r'}_{AB} \hat{\cup} \hat{\Pi}_{A,B}(\mathbf{r}^c)$.

In part 2, if $\hat{\cup}$ were replaced by $\cup$, then the result could be strengthened to equality ($=$) rather than set equality ($\equiv$). However, the merit of stating part 2 as done above will be seen later when we discuss our second rewriting technique. In that setting, it will not matter whether we use equality or set equality and for efficiency purposes not removing duplicates in the union will be an advantage in query evaluation.

3.3 Query Rewriting: Technique I

HV decompositions, in a sense, "expose" structural information about the instance. Our basic idea is to rewrite queries using the decomposition such that structural information is exposed to the DBMS query evaluator. Our thinking is that the optimizer could use this structure to improve query evaluation. Theorem 1, part 1 provides the foundation of our first rewriting technique (illustrated by the following example). Consider an example query, Q:

```
Select Distinct R1.A, R1.B
From R as R1
Where R1.A = 0.
```

Expressed in terms of the bag RA, $Q := \Pi_{A,B}(\sigma_{A=0}(R))$. If the HV decomposition $\mathbf{r}'_{AB}$, $\mathbf{r}'_{AC}$, $\mathbf{r}^c$ is kept in the database, then, by Theorem 1, part 1, we have $Q(\mathbf{r}) = \Pi_{A,B}(\sigma_{A=0}((\mathbf{r}'_{AB} \hat{\bowtie} \mathbf{r}'_{AC}) \hat{\cup} \mathbf{r}^c))$. So, Q can be rewritten as Q_1:

```
Select Distinct R1.A, R1.B
From ((Select RAB.A as A, RAB.B as B, RAC.C as C
      From RAB, RAC
      Where RAB.A=RAC.A)
      Union All
      (Select A,B,C
      From Rc)) as R1
Where R1.A = 0.
```

This technique of query rewriting preserves correctness on any SQL query, *i.e.* the rewritten query is well-formed (no schema conflicts) and, when handed to the DBMS query evaluator, produces exactly the same result as the rewritten query. If R occurs more than once (*e.g.* "R as R2"), then each occurrence of R is replaced as above. We call this *Rewriting Technique I.*

3.4 Query Rewriting: Technique II

In the previous subsection, Q was rewritten as Q_1. However, Q has properties that allow further rewriting. First observe that attribute C does not appear in the output schema of Q and is not used elsewhere in the query. As a result, only the attributes A and B are needed; C can be projected out: $Q(\mathbf{r}) = Q(\hat{\Pi}_{A,B}(\mathbf{r}))$. Hence, we have $Q(\mathbf{r}) = \Pi_{A,B}(\sigma_{A=0}(\hat{\Pi}_{A,B}((\mathbf{r}'_{AB} \hat{\bowtie} \mathbf{r}'_{AC}) \hat{\cup} \mathbf{r}^c)))$.

Now by Theorem 1 part 2, we have $Q(\mathbf{r}) \equiv \Pi_{A,B}(\sigma_{A=0}(\mathbf{r}'_{AB} \hat{\cup} \hat{\Pi}_{A,B}(\mathbf{r}^c)))$. But since $\Pi_{A,B}$ appears at the top level of Q (hence duplicates are removed from the output), then we may replace set equality by equality: $Q(\mathbf{r}) = \Pi_{A,B}(\sigma_{A=0}(\mathbf{r}'_{AB} \hat{\cup} \hat{\Pi}_{A,B}(\mathbf{r}^c)))$. So, Q may be rewritten as Q_2:

```
Select Distinct R1.A, R1.B
```

```
From ((Select RAB.A as A, RAB.B as B
       From RAB)
      Union All
      (Select Rc.A as A, Rc.B as B
       From Rc)) as R1
Where R1.A = 0.
```

The decomposition join has been eliminated and we expect that Q_2, when handed to the DBMS query evaluator, will evaluate faster than Q_1. Moreover, we would expect that Q_2 will evaluate faster than Q if $\mathbf{r}'_{AB}$ and $\mathbf{r}^c$ are not large relative to $\mathbf{r}$. If R occurs more than once in Q, then replace each occurrence as above. We call this *Technique II.*

3.5 Limitations of Technique II

In the previous subsection, Q was rewritten as Q_1 and then further rewritten as Q_2. While rewriting as Q_1 (Technique I) always preserves correctness, rewriting as Q_2 (Technique II) does not. We would like to isolate syntactic properties of Q that guarantee that Technique II preserves correctness. There are two ways in which Technique II may not preserve correctness: the rewritten query is not well-formed (has schema conflicts), the rewritten query does not produce the same output as the original on all inputs.

Since Technique II replaces R by an expression whose schema is $\{A, B\}$, then the original query cannot use C or else the rewritten query may have schema conflicts. The top bag RA expression in Example 1 illustrates how such a schema conflict can arise. Moreover, if the original query involves union or minus, then schema conflicts can also arise. The bottom RA expression in Example 1 illustrates how schema conflicts can arise in the presence of union. Hence, Technique II is not guaranteed to preserve correctness on queries that use C or involve union or minus, because the rewritten query may not be well-defined.

Technique II produces a well-formed query when applied to queries that do not use C and do not contain union or minus. However, the rewritten query may not necessarily produce the same output as the original on all inputs. Clearly, if C is in the output schema of the original query, then the rewritten query may not produce the same output. But, even if C is not in the output schema, the rewritten query may not produce the same output. Consider the following example.

```
Select R1.A, R1.B
From R as R1
Where R1.A = 0.
```

This query is the same as query Q in subsection 3.3 except that the "Distinct" has been removed. Technique II produces a well-defined query and by Theorem 1, part 2, it can be seen that the rewritten query is set equivalent to the original query. However, equality is not preserved. A "Distinct" in the original query is needed to ensure that equality is preserved. Technique II is not guaranteed to preserve correctness on queries which do not have a Distinct at their top level.

Consider yet another example:

```
Select Distinct R1.A
From R as R1
Group by R1.A
Having Count(R1.B) > 2.
```

Here, Technique II produces a well-defined query but does not preserve correctness for reasons similar to the last example. Theorem 1, part 2 implies that replacing R by "((Select * From RAB) Union All (Select A, B From Rc))" preserves set equality, equality is not necessarily preserved. Hence Count may not produce the same result on the original query as the rewrite. Technique II is not guaranteed to preserve correctness on queries involving the aggregate operations count and sum. Aggregate operations max and min do not cause problems, but, for simplicity of presentation, they are omitted from further discussion.

3.5.1 Formal Results

Enough intuition should be developed as to the limitations of Technique II. We give a result (Corollary 1) that defines a general class of queries over which Technique II is guaranteed to preserve correctness. First, though, we state a theorem that defines a general class of bag RA expressions over which Technique II is guaranteed to preserve correctness *up to set equivalence*. Corollary 1 falls out immediately by restricting the class further to those which have Π at their top level.

Let Q be any bag RA expression. Let E be the bag RA expression $R'_{AB}\hat{\cup}\hat{\Pi}_{AB}(R^c)$ where R'_{AB} is a relation symbol over schema $\{A, B\}$ and R^c is a relation symbol over schema $\{A, B, C\}$. Let Q_2 be $Q(R \leftarrow E)$.

Theorem 2 *If Q does not involve union or minus, and the output schema of Q does not contain C, and C does not appear in Q, then Q_2 is well-defined and $Q(\mathbf{r}) \equiv Q_2(\mathbf{r}'_{AB}, \hat{\Pi}_{A,B}(\mathbf{r}^c))$.*

We say that a bag RA expression Q is *top-level distinct* if it is of the form $\Pi_{\ldots}(Q')$. A top-level distinct expression always returns a set.

Definition 1 *We say that a bag RA expression, Q, is* join elimination re-writable (JE-rewritable) *if (i) Q is top-level distinct, (ii) Q does not involve union or minus, (iii) The output schema of Q does not contain C, and (iv) C does not appear in Q (*i.e. *is not part of any projection, selection, or renaming condition, and no join has C among its join attributes).*

Take note that the first example in subsection 3.3 is the SQL of the JE-rewritable, bag RA expression $\Pi_{A,B}(\sigma_{A=0}(\mathbf{r}))$. Theorem 2 implies the following result that defines the class of bag RA expressions over which Technique II preserves correctness.

Corollary 1 *If Q is JE-rewritable, then Q_2 is well-formed and $Q(\mathbf{r}) = Q_2(\mathbf{r}'_{AB}, \hat{\Pi}_{A,B}(\mathbf{r}^c))$.*

The SQL queries equivalent to the JE-rewritable, bag RA expressions are the queries on which Technique II is guaranteed to preserve correctness. We call these *JE-rewritable queries.*

3.6 Hypotheses

We have developed two query rewriting techniques: $Q \rightarrow Q_1$ (Technique I), $Q \rightarrow Q_2$ (Technique II). The first technique is guaranteed to preserve correctness for any query. The second is only guaranteed to preserve correctness for JE-rewritable queries. Q_1 may evaluate faster than Q when handed to the DBMS query evaluator, because Q_1 exposes more of the structure of $\mathbf{r}$, thereby allowing the optimizer to possibly take advantage of this structure. We arrive at our first hypothesis. Let $time(Q)$ denote the time required by the DBMS query evaluator to evaluate Q (likewise, define $time(Q_1)$).

Hypothesis 1 (Rewriting Technique I) *Given query Q, $time(Q_1) < time(Q)$.*

We expect Q_2 to evaluate faster than Q_1 because of the elimination of the join in the HV decomposition. Moreover, we expect Q_2 to evaluate faster than Q when $\mathbf{r}'_{AB}$ and $\mathbf{r}^c$ are small relative to $\mathbf{r}$. Let $|\mathbf{r}|$, $|\mathbf{r}'_{AB}|$, and $|\mathbf{r}^c|$ denote the number of tuples in $\mathbf{r}$, $\mathbf{r}'_{AB}$, and $\mathbf{r}^c$, respectively. Let $adom(A, \mathbf{r}'_{AB})$ denote the active domain of A in $\mathbf{r}'_{AB}$ (likewise define $adom(A, \mathbf{r})$). By definition of HV decompositions, $|\mathbf{r}'_{AB}| = |adom(A, \mathbf{r}'_{AB})|$. We use $\frac{|\mathbf{r}|}{|adom(A,\mathbf{r}'_{AB})|+|\mathbf{r}^c|}$ to quantify the size of $\mathbf{r}'_{AB}$ and $\mathbf{r}^c$ relative to $\mathbf{r}$. For example, if $|adom(A, \mathbf{r}'_{AB})|$ is two percent of $|\mathbf{r}|$ and $|\mathbf{r}^c|$ is twenty percent of $|\mathbf{r}|$, then $|\mathbf{r}|$ is 4.55 times as large as $|\mathbf{r}'_{AB}| + |\mathbf{r}^c|$.

Hypothesis 2 (Rewriting Technique II) *Given JE-rewritable query Q, if $\frac{|\mathbf{r}|}{|adom(A,\mathbf{r}'_{AB})|+|\mathbf{r}^c|} > 1$, then*
$time(Q_2) < time(Q)$. Moreover, if $\frac{|\mathbf{r}|}{|adom(A,\mathbf{r}'_{AB})|+|\mathbf{r}^c|}$ increases, then $time(Q) - time(Q_2)$ also increases.

In the next section, we describe experiments designed to test our hypotheses.

4 Experimental Results

This section describes results from experiments designed to test the hypotheses given earlier. We measure the performance of queries in the context of the decomposition strategy described in the previous section. We first provide details about our method, including a description of the data we used. Next, we report the results from experiments designed to test hypothesis 1 using synthetically generated data. Following that, we test hypothesis 2 using both synthetic and non-synthetic data.

4.1 Experimental Data

Datasets for the experiments were generated randomly, controlling for the size of the relation, the size of the correction, and the size of the active domains for A and B. The order of tuples was permuted to avoid long sequences of tuples in the dataset with the same A and B value. The table used for the join query was generated from $\mathbf{r}$ by projecting out the unique B values and adding a description field, which resulted in the schema $S = \{B, D\}$.

Fixing the size of the active domains provides the benefit of controlling for the selectivity of the queries that select rows based on a constant. The constant used for these queries was the value representing the median frequency. We fixed the size of $adom(B, \mathbf{r})$ to be 100 for each experiment. We considered three sizes for $adom(A, \mathbf{r})$: 100, 1000, and 10000.

For all experiments, the size of the relation was 500,000 tuples. The size of the correction ranged from 0–90% in 10% increments. Note that the 0% correction represents a case where the functional dependency $A \rightarrow B$ holds. The decompositions generated for the experiments were minimal, so $|adom(A, \mathbf{r})| = |adom(A, \mathbf{r'}_{AB})|$ in each case.

The datasets were stored as tables in an Oracle (Version 8.05) database running on Sun UltraSparc 10 server running Solaris 7 equipped with 256 MB of RAM. No indexes were generated for any of the tables. However, statistics were generated for all tables, allowing for cost-based optimization. See Appendix A for a more thorough description of the synthetic data generation procedure.

Queries were executed from a Java 1.3 application using the JDBC thin client interface provided by Oracle. Each query was executed 5 times, with the mean completion time to return the final row in the result reported. The standard deviations we observed were small and are omitted from this report. We used the "NOCACHE" optimizer directive on each query to avoid reusing the cache.[3]

[3]This directive specifies that blocks are placed in the least recently used part of the LRU list in the buffer cache when a full table scan is performed. This was necessary when repeating a query to avoid misleading results.

4.2 Testing Hypothesis 1

We tested the following query:

```
Select Distinct R1.A,R1.B,R1.C
From R as R1
Where R1.A=constant
```

rewritten using Technique I as:

```
Select Distinct R1.A,R1.B,R1.C
From (Select RAB.A,RAB.B,RAC.C
      From RAB,RAC Where RAB.A=RAC.A
       Union All
      Select A,B,C From Rc) as R1
Where R1.A=constant
```

As described in Subsection 3.3, the first query represents Q, while the second query represents Q_1. In the results, we refer to these queries as the original and the rewritten query. Figure 5 depicts the timings we measured for these queries. From these results, it is clear that the rewritten query performs worse than the original query. Consequently, it does not appear that exposing the structure to the optimizer yielded any benefits. Interestingly, the worst performance occurs in the case of a perfect functional dependency. We conclude that our hypothesis is incorrect. We believe that the reason for failure is directly related to the join in the rewritten query. A closer examination of the query plans makes clear why we did not experience an improvement. The plan generated for the rewritten query did not push the selects into the query. Instead, the original relation was materialized and then scanned to get the answer.

4.3 Testing Hypothesis 2

We saw that rewriting Technique I resulted in slower queries due to the cost of carrying out the join introduced by the decomposition. Rewriting Technique II, when applicable, does not

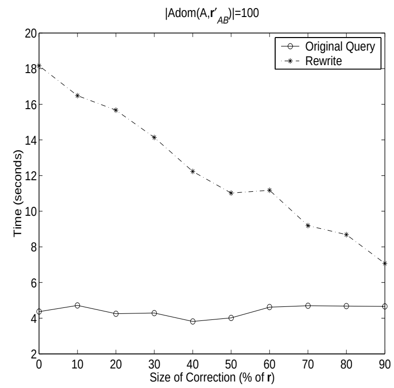

Figure 5: Timings for the query rewritten using Technique I compared to the original query.

produce queries requiring the join. As a result, we expect that the performance of queries rewritten with Technique II will be clearly faster than those rewritten with Technique I. Now we test whether queries rewritten with Technique II are faster than the original queries.

To test hypothesis 2, the following four JE-rewritable queries were used.

1. `Select Distinct R1.A,R1.B From R as R1 Where R1.A=constant`
2. `Select Distinct R1.A,R1.B From R as R1 Where R1.B=constant`
3. `Select Distinct R1.A,R1.B,S.D From R as R1,S Where R1.B=S.B`
4. `Select Distinct R1.A,R1.B From R as R1`

Each query was rewritten as follows:

1. ```
   Select Distinct R1.A,R1.B
   From (Select A,B From RAB Union All Select A,B From Rc) as R1
   Where R1.A=constant
   ```
2. ```
   Select Distinct R1.A,R1.B
   From (Select A,B From RAB Union All Select A,B From Rc) as R1
   Where R1.B=constant
   ```
3. ```
   Select Distinct R1.A,R1.B,S.D
   From (Select A,B From RAB Union All Select A,B From Rc) as R1, S Where R1.B=S.B
   ```
   ```

4. `Select Distinct R1.A,R1.B`

   `From (Select A,B From RAB Union All Select A,B From Rc) as R1`

Figure 6 shows the results for Query 1. In each case, we see that the rewritten queries perform better than the original query when the size of the correction is small. As the size of the correction increases, the performance of the rewritten queries degrades in a linear fashion.

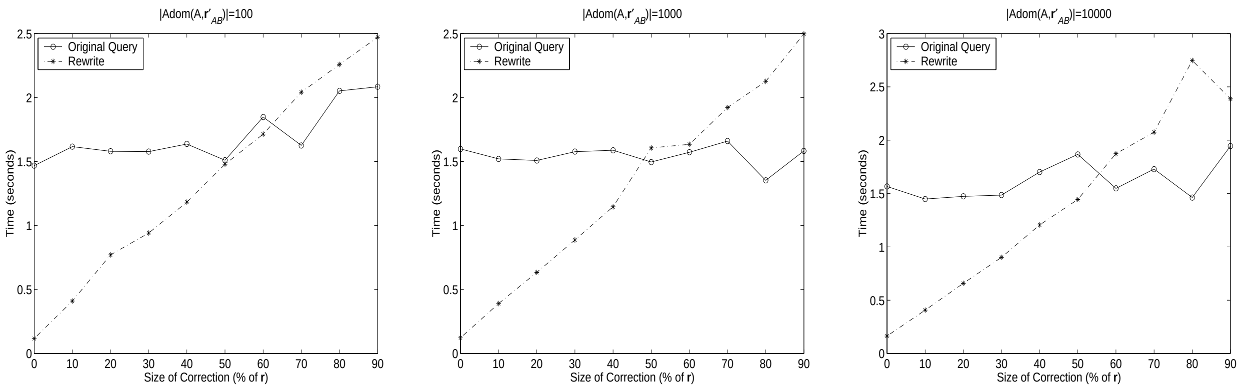

Figure 6: Comparing the mean execution time for Query 1 to the size of the correction.

In Figure 7, we provide the results for Query 2, showing similar behavior to the Query 1 case. As shown in Figure 8 and Figure 9, the performance trend continues. However, since the size of the output for these queries is substantial, the benefits of the decomposition are obscured by the output size. Nevertheless, the rewritten queries continued to outperform the original query for all but the largest correction sizes.

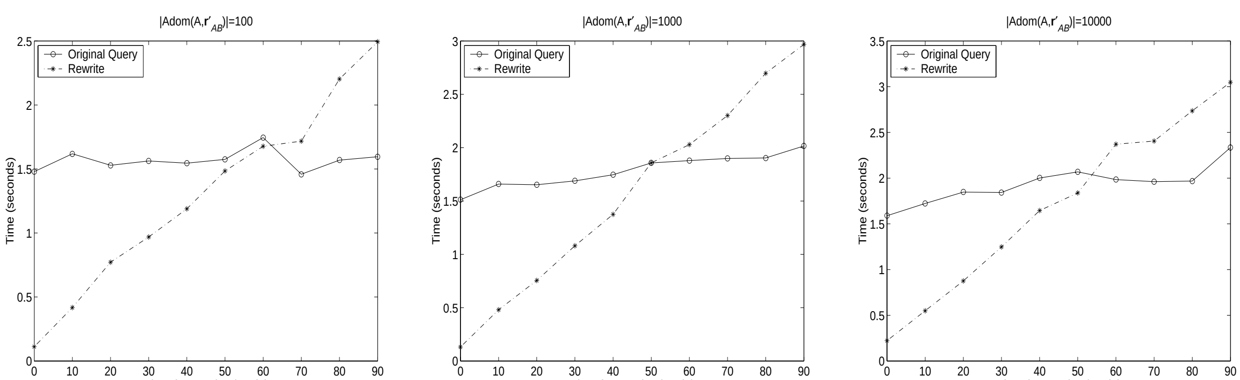

Figure 7: Comparing the mean execution time for Query 2 to the size of the correction.

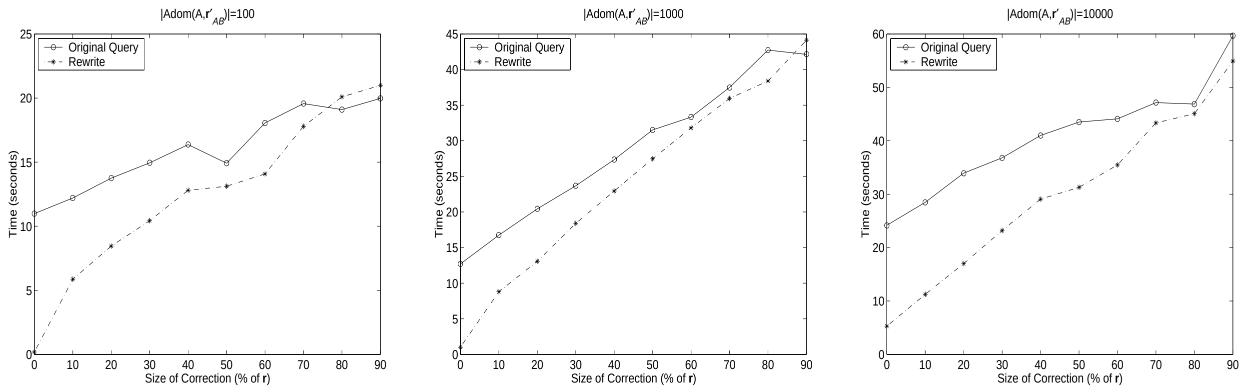

Figure 8: Comparing the mean execution time for Query 3 to the size of the correction.

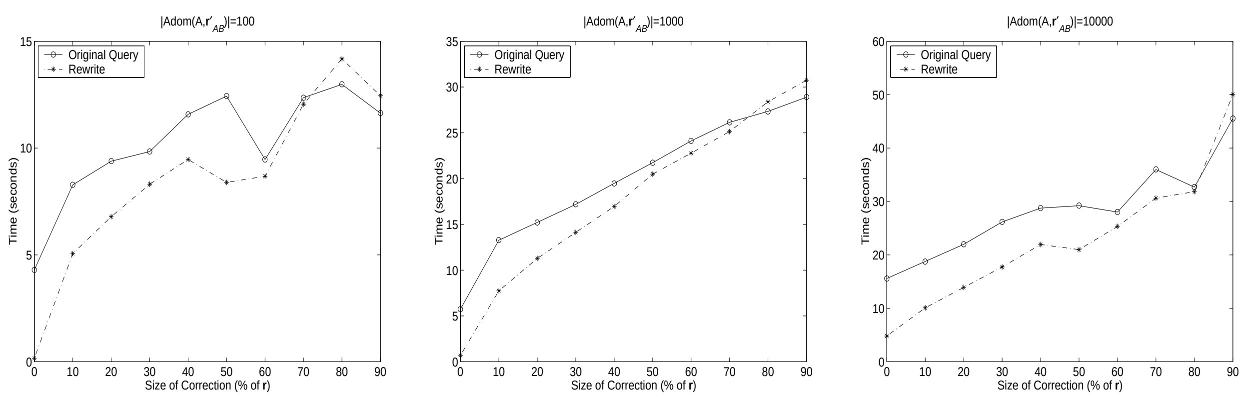

Figure 9: Comparing the mean execution time for Query 4 to the size of the correction.

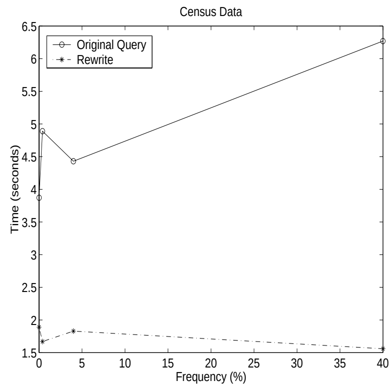

Figure 10: Timings for queries against the census data.

4.4 Census Data

Synthetic datasets are useful for repeated experiments since we can control the dependent characteristics. Real data cannot be controlled in such a way. However, we can decompose the real data in the same way and determine the size of the correction. Then, we can verify that the timings of the queries against the real data are consistent with the synthetic data.

We chose a subset of the attributes contained in the census data, rather than the complete table. The attributes we selected were $industry, meansOfTransportation$, and $maritalStatus$, which correspond to A, B, C in our decomposition. Duplicate tuples were not removed from the projection. The census data we used was the 1990 California 5% sample, with 1.456 million rows. As done with synthetic data, we computed a minimal decomposition. The correction and $Adom(A, \mathbf{r}'_{AB})$ were of size 233,140 (20%) and 245 (¡0.02%), respectively.

we considered was `Select Distinct A,B From Census Where A=constant`. Again, we rewrote the query according to the previously described procedure. We used several constants, based on the frequency of their occurrence in the data. Figure 10 shows these results for 4 frequency cases: High (40%), Low (4%), Very-Low (0.4%) and Non-Existent (0%). For these cases, the rewritten queries all performed better than the original queries. The effect of the frequency of the A value was greater for the original query than for the rewritten query.

The second query we tested against the census data was: `Select Distinct A,B from`

`Census`. The original query executed in an average of 16.7 seconds. In contrast, the rewritten query had an average execution time of 6.9 seconds.

4.5 Discussion

Hypothesis 1 was incorrect due to the cost of the join introduced by the decomposition and the fact that the optimizer did not take advantage of the decomposition. For example, as shown in the queries below, the optimizer could have pushed selects into the decomposition. The first query is the original query rewritten with Technique I used to test hypothesis 1. The second query exploits the structure of the decomposition by pushing selects as deep as possible, which is one of the most basic query optimization strategies. In these experiments the query optimizer did not take advantage of the decomposition.

1. ```
   Select Distinct R1.A,R1.B,R1.C
   From ( Select RAB.A,RAB.B,RAC.C From RAB,RAC Where RAB.A=RAC.A
   Union All Select A,B,C From Rc) as R1
   Where R1.A=constant
   ```

2. ```
   Select Distinct R1.A,R1.B,R1.C
   From (Select RAB.A,RAB.B,RAC.C From RAB,RAC
   Where RAB.A=constant and RAC.A=constant and RAB.A=RAC.A
   Union All Select A,B,C From Rc Where A=constant) as R1
   ```

Queries that were able to use Technique II outperformed the original queries until the size of the correction exceeded 50% - a certainly robust technique. We were pleasantly surprised that the breaking point was so high. After all, it seems intuitive that the decomposition will perform better when the AFD is close to an FD. It is surprising though, that the AFD $A \rightarrow B$ in the original relation can be significantly far from being an FD and result in better performing queries. Observe that for our experimental datasets, the size of the active domain does not materially affect the relative performance of the queries.

When using real data, the results are consistent with our synthetic data when considering the size of the correction. The census data is larger than the synthetic datasets. However, the distribution of values is different for the attributes we selected. In fact the distribution is highly skewed. The rewritten queries perform better against the census data than against the synthetic data. We believe that the skew in the distribution leads to this improvement.

5 Maintaining the Decomposition

So far, we have discussed the HV decomposition without regard to updates to the original relation. In this section we discuss this issue, providing an overview of a strategy to support inserts, updates and deletes.

From the definition of the HV decomposition, there exists minimal decompositions ($\mathbf{r}^c$ is minimal if for all corrections $\mathbf{s}^c$, $|\mathbf{r}^c| \leq |\mathbf{s}^c|$). An update strategy can be used to guarantee minimality when $\mathbf{r}^c$ is initially minimal. We believe, however, that the cost of such a strategy is too high to warrant its use. Instead, our strategy relaxes the minimality constraint.

Upon insertion of a tuple t, an index, for example (preferably a hash) would support constant time lookup in $\mathbf{r}'_{AB}$. The following pseudo-code describes how to handle inserts for a tuple $t = \langle a, b, c \rangle$:

```
if exists (Select * From RAB Where A=a and B=b) Then
    Insert Into RAC Values (a, c)
else
    Insert Into Rc Values (a, b, c)
fi
```

When deleting a tuple, extra work is necessary to handle the case when deleting the last occurrence of an a from $\mathbf{r}'_{AC}$, which forces a tuple to be deleted from $\mathbf{r}'_{AB}$. Ideally, the best performance for the delete will result when an index on $\mathbf{r}'_{AC}$ exists in addition to the index on $\mathbf{r}'_{AB}$.

Given $t = \langle a, b, c \rangle$ to delete:

```
if exists (Select * From RAB Where A=a and B=b) Then
    Delete from RAC Where A=a and C=c
```

```
    if not exists (Select * From RAC Where A=a) Then
        Delete from RAB Where A=a and B=b
    fi
else
    Delete from Rc Where A=a and B=b and C=c
fi
```

Updates to existing tuples are handled by a delete followed by an insert. Since these operations may yield a non-minimal decomposition, then it may be necessary to reorganize the decomposition to restore minimality (if desired). This is probably best done as a scheduled, batch process. As described in Subsection 4.3, the size of $\mathbf{r}^c$ can be significantly large while still enjoying performance benefits, which indicates that a large number of insert or delete operations can occur before reorganization is required.

6 Conclusion and Future Work

In this paper we investigated an approach (paralleling recent work extending SQO) to improve query evaluation. Our approach is based on decomposing relation instances with respect to AFDs and rewriting queries to take advantage of these decompositions (HV decompositions). The primary idea is that the semantic information contained in an AFD can be exposed by creating an HV decomposition of the relation instance. This information can then be exploited to speed up query evaluation: rewrite the query to use the decomposition instead of the original relation, then, issue the rewritten query to the DBMS query engine instead of the original query. This process is expressed pictorially in Figure 1. Two things in particular should be pointed out about how our approach fits into the literature. First, it represents (to our knowledge) the first use of AFDs in query evaluation. Second, it addresses the question raised by the work of Godfrey *et al.* described in Section 2. Our primary motivation was not to specifically address the question. We discovered after we had obtained our results that they could be used to address the question.

We investigated two rewriting techniques. Technique I replaces all occurrences of the

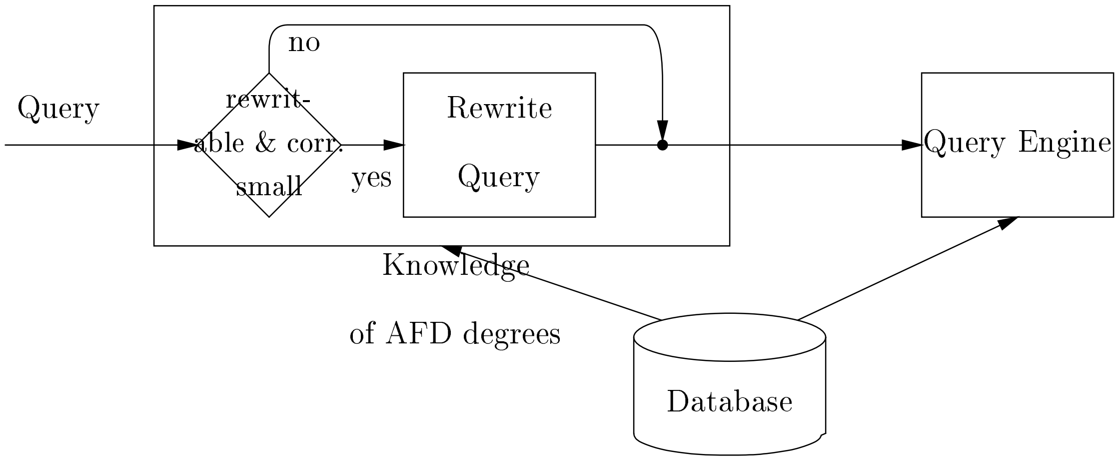

Figure 11: The preprocessor rewrites JE-rewritable queries provided that the correction is not too large.

relation symbol by its decomposition. This technique is guaranteed to preserve correctness on all SQL queries. The motivation was that the optimizer can take advantage of the decomposition and produce a more efficient plan. Our experiments, however, point out that this was not the case. The introduction of the decomposition join caused the rewritten queries to run more slowly. Technique II replaces all occurrences of the relation symbol by the decomposition without the join. This technique is only guaranteed to preserve correctness on a special class of queries (JE-rewritable queries). However, our experiments show that queries rewritten with this technique tend to evaluate significantly faster than the original query, provided the correction was not too large.

Our results suggest how the architecture depicted in Figure 1 can be sharpened. We have observed that Technique II can offer significant speed-up, but, it can only be applied to JE-rewritable queries. So, the original relation should be kept along with its HV decomposition. The preprocessor examines the query to see if it is JE-rewritable. If so, and if the correction is not too large, then the query is rewritten. Otherwise, the query is not rewritten and the original query is handed to the DBMS query engine. See Figure 11.

There are a number of directions for future work. 1. Address the primary drawback to Technique II: it is guaranteed to preserve correctness only on JE-rewritable queries. In particular, modify our approach to handle aggregate operations like count and sum. This can be achieved by modifying the construction of $\mathbf{r}'_{AB}$ to keep counts. The significant issue is then sharpened: how to rewrite queries to use the counts. 2. Investigate other rewritings in the AFD context. For example, an FD can allow elimination of outer join and group by operations. 3. Investigate the use of other decompositions. A natural candidate is a decomposition induced by approximate multi-valued dependencies, which creates two correction relations $\mathbf{r}^+, \mathbf{r}^-$. 4. Investigate methods for maintaining an HV decomposition. Does the extra time required for processing inserts and deletes eclipse the gains in query evaluation? Also, when should corrections be reorganized to minimize $\mathbf{r}^c$? The $\mathbf{r}^+$, $\mathbf{r}^-$ decomposition mentioned above is intriguing because deletes are processed via $\mathbf{r}^-$.

In closing we point out that the primary purpose was to introduce the idea of exploiting AFDs in query evaluation via HV decompositions and demonstrate that the idea is fertile grounds for future work. We feel that we have achieved this goal.

References

[1] Bell S. Deciding distinctiveness of query results by discovered constraints. *Proc. of the 2nd International Conf. on the Practical Application of Constraint Technology* (1996), 399–417.

[2] Cavallo R. and Pittarelli M. The theory of probabilistic databases. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)* (1987), pp. 71–81.

[3] Chakravarthy U., Grant J., and Minker J. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems 15*, 2 (June 1990), 162–207.

[4] Cheng Q., Gryz J., Koo F., Leung C., Liu L., Qian X., and Schiefer B. Implementation of two semantic query optimization techniques in DB2 UDB. In *Proc. of the 25th Conf. on Very Large Data Bases* (Edinburgh, Scotland, 1999), pp. 687–698.

[5] Dalkilic M. and Robertson E. Information dependencies. In *Proc. of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (2000), ACM, pp. 245–253.

[6] Godfrey P., Grant J., Gryz J., and Minker J. *Logics for Databases and Information Systems.* Kluwer Academic Publishers, Boston, MA, 1998, pp. 265–307.

[7] Godfrey P., Gryz J., and Zuzarte C. Exploiting constraint-like data characterizations in query optimization. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data* (May 2001), pp. 582–592.

[8] Goodman L. and Kruskal W. Measures of associations for cross classifications. *Journal of the American Statistical Association 49* (1954), 732–764.

[9] Hammer M. and Zdonik S. Jr. Knowledge-based query processing. In *Proc. of the Sixth Intl. Conf. on Very Large Data Bases* (Montreal, Canada, Oct. 1980), pp. 137–147.

[10] Hsu C. and Knoblock C. Using inductive learning to generate rules for semantic query optimization. In *Advances in Knowledge Discovery and Data Mining* (1996), Fayyad U. and Piatetsky-Shapiro G., Ed.

[11] Huhtala Y., Kärkkäinen J., Porkka P., and Toivonen H. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings 14th International Conference on Data Engineering* (February 1998), IEEE Computer Society Press, pp. 392–401.

[12] Jarke J., Clifford J., and Vassiliou Y. An optimizing PROLOG front-end to a relational query system. In *Proc. of the ACM SIGMOD Conf.* (1984), pp. 296–306.

[13] Kantola M., Mannila H., Räihä K., and Siirtola H. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems 7* (1992), 591–607.

[14] King J. QUIST–a system for semantic query optimization in relational databases. In *Proc. of the 7th International Conf. on Very Large Data Bases Cannes* (Cannes, France, Sept. 1981), IEEE Computer Society Press, pp. 510–517.

[15] King J. Reasoning about access to knowledge. In *Proc. of the Workship on Data Abstraction, Databases, and Conceptual Modelling* (Jan. 1981), SIGPLAN Notices, pp. 138–140.

[16] Kivinen J. and Mannila H. Approximate dependency inference from relations. *Theoretical Computer Science 149*, 1 (1995), 129–149.

[17] Lee T. An information-theoretic analysis of relational databases–part I: Data dependencies and information metric. *IEEE Transactions on Software Engineering SE-13*, 10 (October 1987), 1049–1061.

[18] Lopes S., Petit J., and Lakhal L. Efficient discovery of functional dependencies and Armstrong relations. In *Lecture Notes in Computer Science 1777 (first appeared in the Proceedings of the Seventh International Conference on Extending Database Technology (EDBT))* (2000), pp. 350–364.

[19] Malvestuto F. Theory of random observables in relational databases. *Information Systems 8(4)* (1983), 281–289.

[20] Mannila H. and Räihä K. Dependency inference. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)* (1987), pp. 155–158.

[21] Mannila H. and Räihä K. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering 12* (1994), 83–99.

[22] Nambiar K. Some analytic tools for the design of relational database systems. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)* (1980), pp. 417–428.

[23] Novelli N., Cicchetti R. FUN: an efficient algorithm for mining functional and embedded dependencies. In *Lecture Notes in Computer Science 1973 (Proceedings of the 8th International Conference on Database Theory (ICDT))* (2001), pp. 189–203.

[24] Paulley G. *Exploiting Functional Dependence in Query Optimization.* PhD thesis, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Sept. 2000.

[25] Paulley G. and Larson P. Exploiting uniqueness in query optimization. In *Proc. of the 10th ICDE* (1994), pp. 68–79.

[26] Piatetsky-Shapiro G. Probabilistic data dependencies. In *Machine Discovery Workshop (Aberdeen, Scotland)* (1992).

[27] Pirahesh H., Hellerstein J., and Hasan W. Extensible/rule based query rewrite optimization in starburst. In *Proc. 1992 ACM-SIGMOD Int. Conf. Management of Data* (May 1992), pp. 39–48.

[28] Pirahesh H., Leung T.Y., and Hasan W. A rule engine for query transformation in starburst and ibm db2 c/s dbms. In *Proc. 13th Int. Conf. on Data Engineering (ICDE)* (April 1997), pp. 391–400.

[29] Ramakrishanan R. and Gehrke J. *Database Management Systems 2nd Edition.* McGraw-Hill Higher Education, Boston, MA, 2000.

[30] Sagiv Y. Quadratic algorithms for minimizing join in restricted relational expressions. *SIAM Journal of Computing 12*, 2 (May 1983), 316–328.

[31] Shekar S., Srivastava J., and Dutta S. A formal model of trade-off of a semantic query optimizer. In *Proc. of 14th on Very Large Data Bases Los Angeles* (1988), pp. 457–467.

[32] Shekhar S., Hamidzadeh B., Kohli A., and Coyle M. Learning transformation rules for semantic query optimization: a data-driven approach. *IEEE Transactions on Knowledge and Data Engineering 5*, 6 (December 1993), 950–964.

[33] Shenoy S. and Ozsoyglu Z. A system for semantic query optimization. In *Proc. 1987 ACM-SIGMOD Int. Conf. Management of Data* (May 1987), pp. 181–195.

[34] Shenoy S. and Ozsoyglu Z. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering 1*, 3 (Sept. 1989), 344–361.

[35] Silberschatz A., Korth H., and Sudarshan S. *Database System Concepts.* McGraw-Hill Companies, Inc., New York, 1997, pp. 193–214.

[36] Stonebraker M. Implementation of integrity constraints and view by query modification. In *Proc. of the ACM SIGMOD Conf.* (May 1975), ACM, pp. 65–78.

[37] Sun W. and Yu C. Semantic query optimization for tree and chain queries. *IEEE Transactions on Knowledge and Data Engineering 6*, 1 (February 1994), 136–151.

[38] Wyss C., Giannella C., and Robertson E. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *Lecture Notes in Computer Science 2114 (Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery)* (2001), pp. 101–110.

[39] Yu C. and Sun W. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering 1*, 3 (September 1989), 362–374.

Appendix A

In this section we provide details on the procedure and algorithm we used to generate synthetic data for our experiments. Pseudo-code for the algorithm is provided, as well as descriptions of the various control variables used.

The goal for the procedure is to generate relations $\mathbf{r}, \mathbf{r}'_{AB}, \mathbf{r}'_{AC}$, and $\mathbf{r}^c$, as described previously. The following variables are used to control the generation:

Variable	Description
`sizeOfR`	The number of tuples in $\mathbf{r}$.
`sizeOfRc`	The number of tuples in $\mathbf{r}^c$.
`sizeOfA`	The size of $Adom(A, \mathbf{r})$.
`sizeOfB`	The size of $Dom(B, \mathbf{r})$.
`maxRepeats`	The maximum number of consecutive (a, b) values.

Note that `sizeOfB` represents the maximum number of possible B values in $\mathbf{r}$, which differs from `sizeOfA` - the actual number of A values. The algorithm generates both $\mathbf{r}$ and the decomposition in the same pass. The data generation algorithm is shown below:

```
procedure Generate(sizeOfR, sizeOfRc, sizeOfA, sizeOfB, maxRepeats) :

  // create the possible a and b values
  A[1..sizeOfA] = unique random values between 1 and sizeOfR;
  B[1..sizeOfB] = random values between 1 and sizeOfR;

  // create the a,b functional dependencies
  for i = 1 to sizeOfA
    AB[i] = random value taken from B;
    insert into RAB values (A[i], AB[i]);

  // create the tuples in R and Rc and Rac
  currentTuple := 0;
  currentA := 1;
  while currentTuple < sizeOfR

    repeat := random value between 1 and maxRepeats;

    if (currentTuple >= (sizeOfR - sizeOfRc)
      b := random value taken from B such that b != AB[currentA]

    for i = 1 to repeat
      if (currentTuple >= (sizeOfR - sizeOfRc) then
        // generate the tuples in R and Rc
        insert into R values (A[currentA], b, currentTuple);
        insert into Rc values (A[currentA], b, currentTuple);
      else
        // otherwise generate the FD tuples in R and RAC
        insert into R values (A[currentA], AB[currentA], currentTuple);
        insert into RAC values (A[currentA], currentTuple);
```

```
      fi
      currentTuple++;
    end for;

    // cycle through all the possible A values
    currentA++;
    if currentA > sizeOfA then
      currentA := 1;

  end while;

end Generate;
```