OLAP over Imprecise Data with Domain Constraints

Doug Burdick[1], **AnHai Doan**[1], **Raghu Ramakrishnan**[2], **Shivakumar Vaithyanathan**[3]
[1]University of Wisconsin - Madison, [2]Yahoo! Research, [3]IBM Almaden Research Center

ABSTRACT

Several recent papers have focused on OLAP over imprecise data, where each fact can be a region, instead of a point, in a multi-dimensional space. They have provided a multiple-world semantics for such data, and developed efficient ways to answer OLAP aggregation queries over the imprecise facts. These solutions, however, assume that the imprecise facts can be interpreted *independently* of one another, a key assumption that is often violated in practice. Indeed, imprecise facts in real-world applications are often correlated, and such correlations can be captured as domain integrity constraints (e.g., repairs with the same customer names and models took place in the same city, or a text span can refer to a person or a city, but not both).

In this paper we provide a framework for answering OLAP aggregation queries over imprecise data in the presence of such domain constraints. We first describe a relatively simple yet powerful constraint language, and formalize what it means to take into account such constraints in query answering. Next, we prove that OLAP queries can be answered efficiently given a database $D*$ of fact marginals. We then exploit the regularities in the constraint space (captured in a constraint hypergraph) and the fact space to efficiently construct $D*$. We present extensive experiments over real-world and synthetic data to demonstrate the effectiveness of our approach.

1. INTRODUCTION

OLAP employs a multi-dimensional data model, where each fact can be viewed as a *point* in the corresponding multi-dimensional space. If we relax the assumption that all facts are points, and allow some facts to be *regions*, we must handle the resulting imprecision when answering queries. For example, we can denote that a particular auto repair took place in the state of Wisconsin, without specifying a city. Answering queries over such imprecise information is widely recognized as important and has received increasing attention. In particular, we have recently developed an efficient solution [8, 7], which provides a possible-world interpretation for imprecise facts, then computes the result of an aggregation query Q to be the expected value of evaluating Q over all possible worlds.

A key assumption underlying the above solution is that the imprecise facts are *independent* [8]. This assumption is often violated in practice, as the following example demonstrates:

FactID	*Loc*	*Auto*	*Name*	*Cost*
p_1	WI	F150	John Smith	100
p_2	WI	F150	John Smith	250
p_3	Madison	Honda	Dells	130
p_4	Dells	Honda	Madison	130

Table 1: A sample automotive repair database.

EXAMPLE 1. *Consider the automotive repair database in Table 1, where each tuple describes a repair. Here, both facts p_1 and p_2 are imprecise, because they do not specify the particular city in Wisconsin (e.g., Madison, Dells, or perhaps Milwaukee) where the repair took place. Now suppose that we wish to exploit the* domain knowledge *(something that we independently know to be true in the real world, or a strong belief that we want to impose on the possible worlds considered in answering the query) that all repairs with the same customer name and the same auto model take place in the same city. Then facts p_1 and p_2 are not independent. For instance, if in a particular world the repair of fact p_1 took place in Madison, then so did the repair of p_2.*

As another example, suppose the facts of Table 1 are extracted from text documents that describe repairs. In particular, consider the text snippet "Madison, Honda, broken ex. pipe, Dells & I-90, towed 25 miles, $130". *A reasonable person-name extractor may extract "Madison" and "Dells" as person names, and similarly a reasonable location extractor may extract the same "Madison" and "Dells" as location names. This results in the two facts p_3 and p_4 in Table 1, which reflect different interpretations of "Madison" and "Dells". However, we know that each text span can have only one interpretation (e.g., either person name or location, but not both). Consequently, facts p_3 and p_4 are not independent. In particular, if we accept p_3 then we must eliminate p_4 from the fact database, and vice versa.*

The above examples show that in OLAP over imprecise data, we often have extra information about which combinations of completions of facts are possible. This extra information either reflects the application logic (e.g., repairs with the same customer names and models took place in the same city), or the logic of the fact-derivation process (e.g., if facts are extracted from text then a text span encodes only a single interpretation). A natural way to interpret this information is as *constraints* over the set of possible worlds. Then, given a set of such constraints, our problem is to answer OLAP queries over *only the subset of the worlds that satisfy the constraints*.

This paper proposes an efficient solution to the above problem. In what follows, we will first describe our prior framework on

VLDB '07, September 23-28, 2007, Vienna, Austria.

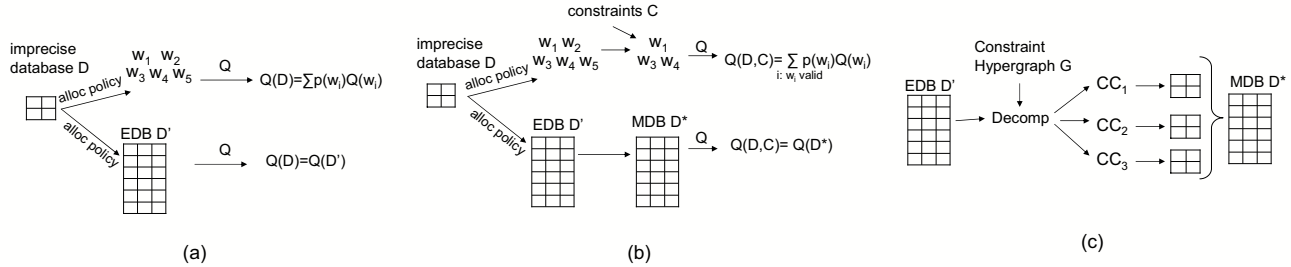


Figure 1: (a) Querying imprecise data in prior work, (b)-(c) exploiting domain constraints for querying imprecise data in this work.

OLAP over imprecise data [8, 7], then the challenges of pushing constraints into this framework and our solutions.

Figure 1.a illustrates our prior framework. Let D be a database of imprecise facts. A popular semantics [24, 19] interprets D as a set of possible worlds, such as $w_1 - w_5$ as shown in the figure, where each world is created by selecting a possible completion for each fact. Now consider an aggregation query Q over D. The answer to Q, denoted as $Q(D)$, is taken to be the expected answer of Q over all possible worlds ($w_1 - w_5$ in this case).

Computing $Q(D)$ by evaluating Q over *all* possible worlds is practically infeasible. Hence, the work [8] proposes an efficient solution to this problem. First, it shows how the set of possible worlds can be compactly encoded using an *extended database* (*EDB* for short) D', and how to create D' from D using an *allocation policy* (see Figure 1.a). Next, it shows that query Q can be evaluated quickly in a single scan over D', and the answer, denoted $Q(D')$, is the same as $Q(D)$. This work is followed up in [7] by developing efficient allocation algorithms to compute D' from D.

In this paper, we significantly extend the above framework (see Figures 1.b-c). We begin by defining a language to represent domain constraints. Next, we modify the query-answering semantics to exploit such constraints. Consider again the five worlds $w_1 - w_5$ that result from the database D of imprecise facts. Given a set of constraints C, we retain only the *valid worlds*, i.e., those that satisfy C (which are w_1, w_3, w_4 in this case, see Figure 1.b). Then we define the answer of Q over the imprecise database D and the set of constraints C, denoted $Q(D, C)$, to be the expected answer of Q over the valid worlds (Figure 1.b).

We then develop an efficient way to compute $Q(D, C)$, without enumerating all valid worlds. This is the central technical challenge we address in this paper. Clearly, we cannot answer Q over EDB D', as in prior work, because that would violate our semantics of considering only valid worlds. Instead, we prove that if we can construct a *marginal database* (or *MDB* for short) $D*$, which assigns to each fact completion its marginal probability in the valid worlds, then we can answer $Q(D, C)$ efficiently in a single scan over MDB $D*$ (Figure 1.b, the lower half). This result is surprising, because in many problem settings with domain constraints [19], even the existence of a MDB $D*$ does not help compute exact query answers (and often approximate solutions are proposed instead [19]). We show that for the algebraic aggregation operators [16] commonly used in OLAP queries (e.g., Sum, Count, approximate Average), it is possible to compute an exact answer using the MDB.

We then turn our attention to the problem of efficiently constructing MDB $D*$, given an EDB D' and a set of constraints C. To solve this problem, first we create a hypergraph G that captures the regularities in the constraint space (see Figure 1.c). Next, we exploit these regularities, and use G to decompose D' into independent *connected components* (e.g., $CC_1 - CC_3$, as shown in Figure 1.c). To ensure efficient decomposition, we store both the EDB D' and the constraint hypergraph G in a RDBMS and execute the decomposition using SQL queries. We next process each component in isolation to generate a portion of the MDB database $D*$, then combine these portions to obtain the final $D*$.

Processing each component is in itself a difficult problem. Even though each component CC_i tends to be far smaller than the original EDB D', it is still often large enough to make exhaustive processing impractical. To address this problem, we employ a technique called *variable elimination* in the probabilistic inference literature [20]. The key idea is to we exploit regularities in both the constraints and fact space, to fully complete certain imprecise facts, which are "bottleneck" variables in the component. This breaks the component into smaller independent "chunks" that now can be easily processed in isolation.

To summarize, this paper makes the following contributions:

- We describe a simple yet powerful language to model domain constraints, then define the semantics of query answering in OLAP over imprecise data in the presence of such constraints.
- In the above setting, we prove that a database of fact marginals can be used to answer OLAP aggregation queries efficiently.
- We develop an algorithm to decompose an imprecise database into independent components, exploiting regularities in the constraint space, as well as relational optimization techniques.
- We develop an algorithm that can process each component efficiently, by exploiting regularities in both the constraints and the fact space. Taken together, our algorithms enable us to quickly compute the database of fact marginals.
- We present extensive experiments to demonstrate that our algorithms scale up to large data sets and complex constraints.

An extended version of this paper with all proofs can be found in [9]. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides background and introduces our notation. Sections 4-6 describe our solution. Section 7 presents experimental results, and Section 8 concludes.

2. RELATED WORK

This is the first work we are aware of that addresses the issues of performing OLAP aggregation over imprecise and uncertain data with constraints. There is much work that separately addresses each of these issues (OLAP aggregation, querying imprecise data, and answering queries with constraints), which we will not attempt to summarize here. Overviews of several different topics in OLAP aggregation can be found in [10, 27].

Although constraints have been considered in the OLAP setting, these constraints either addressed data modeling [21, 26] or were constraints over query results over precise data [28]. The work on constraints in OLAP has not considered imprecise data. Although there has been much work addressing uncertain and imprecise data (e.g., [22, 30, 33, 32, 4, 17, 12, 29]), this work has not addressed

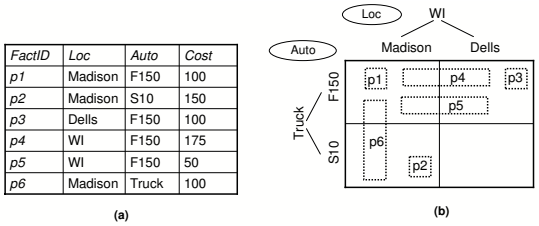


Figure 2: (a) A sample fact table D, (b) a multi-dimensional view of facts in D

answering OLAP aggregation queries over such data. The very recent work by [4] presents an approach for efficiently representing and querying arbitrary sets of possible worlds; however, they do not consider aggregation queries.

Our work in [8] is the first work we are aware of to address semantic issues specific to OLAP aggregation queries over imprecise and uncertain data, namely maintaining consistency between sets of related queries (e.g, roll-up and drill-down). The related work sections in [8, 7] provides further details how that work (and this current work as well) is distinct from prior related work involving aggregation queries over imprecise data. However, neither [8, 7] nor any of the related work described in these consider OLAP aggregation over imprecise data *in the presence of constraints*.

While there has been considerable work on querying inconsistent databases [5, 3, 11, 4, 6, 1, 15, 25], there has been relatively little work addressing aggregation queries over such data. Compared to this work, we make two significant contributions. First, the constraint language we consider is significantly more general than ones considered in most prior work. Typically, these prior works only address constraints in the form of functional dependencies. A notable exception is [15], which considers a language with aggregate constraints. However, this work does not address imprecise data. Second, we consider *OLAP* aggregation queries, while prior work in [5, 25] only addresses *scalar* aggregation queries and do not address the additional consistency requirements present in the OLAP setting [8].

3. PRELIMINARIES

We now review the framework in [8] for OLAP over imprecise and uncertain data. Later we significantly extend this framework to handle domain integrity constraints.

Data Representation: Standard OLAP considers two types of attributes: dimensions (e.g., *Location*) and measures (e.g.,*Cost*). In prior work [8] we have extended this model to support imprecision in dimension values. Specifically, each dimension in standard OLAP takes value from a base domain B (e.g., location takes value from $B = \{$`Madison,Dells`$\}$). In the extended model, each dimension takes value from an *hierarchical domain* H over B.

DEFINITION 1 (HIERARCHICAL DOMAINS). *A* hierarchical *domain H over base domain B is a power set of B such that (a) $\emptyset \notin H$, (b) H contains every singleton set (i.e., corresponds to some element of B), and (c) the values of H either subsumes one another or are disjoint, i.e., $\forall h_1, h_2 \in H$, $h_1 \supseteq h_2$, $h_1 \subseteq h_2$, or $h_1 \cap h_2 = \emptyset$. Non-singleton elements of H are called* imprecise *values.*

We then define a *fact table schema* to be of the form $\langle A_1, \ldots, A_k, M_1, \ldots, M_n \rangle$, where each dimension attribute A_i has an associated domain $dom(A_i)$ that is *imprecise*, and each measure attribute M_j has an associated domain $dom(M_j)$ that is numeric. Next, we define

FactID	*Loc*	*Auto*	*Cost*	$p_{c,r}$
p_1	Madison	F150	100	1
p_2	Madison	S10	150	1
p_3	Dells	F150	100	1
p_4	Madison	F150	175	0.6
p_4	Dells	F150	175	0.4
p_5	Madison	F150	50	0.7
p_5	Dells	F150	50	0.3
p_6	Madison	F150	100	0.5
p_6	Madison	S10	100	0.5

Table 2: An EDB D' for the database of facts D in Figure 2.a.

DEFINITION 2 (FACT AND FACT TABLE). *A fact table D is a collection of facts of the form $\langle a_1, \ldots, a_k, m_1, \ldots, m_n \rangle$ where $a_i \in dom(A_i)$ and $m_j \in dom(M_j)$. In particular, if dom(A_i) is hierarchical, a_i can be any leaf or non-leaf node in dom(A_i). We will use the terms "fact table" and "imprecise database" interchangeably, when there is no ambiguity.*

Intuitively, such a fact $r = \langle a_1, \ldots, a_k, m_1, \ldots, m_n \rangle$ maps into a region $reg(r)$ in the k-dimensional space S formed by the dimension attributes $A_1, \ldots, A_k$. To formalize this notion, we defined a *cell* in S to be a vector $\langle c_1, \ldots, c_k \rangle$ such that every c_i is an element of the base domain of A_i. The region $reg(r)$ is then the set of cells $\{\langle c_1, \ldots, c_k \rangle \mid c_i \in a_i, i \in 1 \ldots k\}$. Each cell in $reg(r)$ represents a possible *completion* of fact r, formed by replacing all non-leaf node a_i with a leaf node from the subtree rooted at a_i.

EXAMPLE 2. *Figure 2.a shows a fact table with two dimension attributes:* Loc *and* Auto*, and one measure attribute:* Cost*. The dimensions take values from their associated hierarchical domains. Figure 2.b shows the structure of these domains and the regions of the facts.*

Facts $p_1 - p_3$ are precise in that their dimension attributes take leaf-node values. Therefore they map to individual cells in Figure 2.b. Facts $p_4 - p_6$ on the other hand are imprecise and map to the appropriate two-dimensional regions. For example, fact p_6 is imprecise because dimension Auto *takes the non-leaf node value* `Truck`*. The region of p_6 consists of cells* `(Madison,F150)` *and* `(Madison,S10)`*, which represent possible claims of p_6.*

Possible-World Semantics for Querying: Next, we defined a possible-world semantics for imprecise facts as follows. Let D be a database of facts (i.e., a fact table, as defined earlier). As discussed earlier, completing an imprecise fact $r \in D$ means "assigning" r to a cell $c \in reg(r)$, thereby eliminating the imprecision of r. By completing *all* imprecise facts in D, we obtain a database W that contains only precise facts. We call W a *possible world* for D, and the multiple choices for completion (for each fact) clearly lead to a set of possible worlds for D.

To formalize the above process, in [8] we first defined the notion of *allocation*: given a fact r, the *allocation* of r to a cell $c \in reg(r)$ is a non-negative quantity $p_{c,r}$, called *allocation weight*, which denotes the weight of completing r to cell c, such that $\sum_{c \in reg(r)} p_{c,r} = 1$.

Next, we defined an *allocation policy* to be a procedure that inputs a fact table D and outputs a table D' that consists of the allocations of *all* facts in D. We now can define

DEFINITION 3 (CLAIM AND EXTENDED DATABASE EDB). *Let D be a fact table. For each fact $r \in D$, an allocation policy creates a set of tuples $\{\langle id(r), c, p_{c,r} \rangle | c \in reg(r), p_{c,r} > 0, \sum p_{c,r} = 1\}$, where $id(r)$ is the id of r. Observe that each precise fact has a single allocation of 1 for the cell to which it maps.*

We call each tuple $\langle id(r), c, p_{c,r} \rangle$ a claim, and the collections of all such claims an extended database D', *or* EDB *for short.*
Intuitively, each claim for an imprecise fact r corresponds to a possible completion of r. In this work, we use the terms completion and claim interchangeably. In [8, 7] we introduced several important allocation policies, and showed how to efficiently execute them over a fact table D to generate the database EDB D'.

Now if we select from the EDB D' a set of claims that correspond to one claim per fact, then we obtain a possible world W for D. Furthermore, we compute the *probability* of the world W to be the product of the allocation weights of all selected claim tuples. (See [8] for a motivation of this procedure.)

EXAMPLE 3. *Table 2 shows a possible EDB D' for the database of facts D in Figure 2.a. Here, attribute* Loc = `WI` *of fact p_4 can complete to either* `Madison` *or* `Dells`*, thus creating the two claims with id p_4 in the table. These completions have probabilities 0.6 and 0.4, respectively (see column $p_{c,r}$ of the table). Suppose we select the very first completion for each of the facts $p_1 - p_6$. Then we obtain a world W with probability* $1 \cdot 1 \cdot 1 \cdot 0.6 \cdot 0.7 \cdot 0.5 = 0.21$.

Thus, given any database of fact D and an allocation policy A, the resulting EDB D' conceptually defines a set of all possible worlds $W_1, \ldots, W_m$, together with probabilities $p_1, \ldots, p_m$, respectively. We then defined the result of an OLAP query Q over D to be the expected value of Q over the worlds $W_1, \ldots, W_m$. For instance, if Q is Sum and its answers for $W_1, \ldots, W_m$ are $v_1, \ldots, v_m$, respectively, then its answer for D is $\sum_{i=1}^{m} p_i * v_i$.

In [8] we then demonstrated that the answer computed using the above possible-world semantics have several desirable properties (e.g., consistency and faithfulness). Finally, we showed how to compute such an answer efficiently, via a single scan of the EDB D' (thus avoiding the expensive process of enumerating all possible worlds). See [8] for more details.

4. CONSTRAINT LANGUAGE

We now describe a relatively simple yet powerful language to specify domain constraints over the imprecise data. Sections 5-6 show how to execute OLAP queries in the presence of such constraints.

4.1 Syntax

We begin by defining the notion of atom, which we then use later to define constraints:

DEFINITION 4 (ATOMS). *Let D be a fact table (see Definition 2), an* atom *is of the form $[r.A\,\theta\,c]$, $[r.A\,\theta\,r'.A]$, $exists(r)$, or $\neg exists(r)$ where:*

- *r, r' are either variables that bind to factIDs in D or specific factIDs themselves, and $r.A$ is the value of (dimension or measure) attribute A of fact r;*
- *$\theta \in \{=, \leq, \geq, <, >\}$ is a comparison operator over the appropriate domain;*
- *c is a constant from $dom(A)$; and*
- *$exists(r)$ ($\neg exists(r)$) is a predicate that holds if fact r exists (cannot exist).*

Note that in the above definition, constant c is from $dom(A)$, and hence can be either precise (e.g., `Madison`) or imprecise (e.g., `WI`). The operators in θ can be the domain-specific version of the comparison operators listed. For example, for dimensions extracted from text "=" may be implemented as a string comparison routine. The only requirement placed on θ is that each atom must evaluate to logical true or false. We now can define constraints as follows:

DEFINITION 5 (CONSTRAINTS). *A* constraint *is an implication of the form $\Phi_1 \Rightarrow \Phi_2$, where Φ_1, Φ_2 are conjunctions of atoms (i.e., $\wedge_i atom_i$).*

EXAMPLE 4. *Using the above language, we can write the first constraint introduced in Example 1, "repairs with the same customer names and models took place in the same city", as follows:*

$$(r.Name = r'.Name) \quad \wedge \quad (r.Auto = r'.Auto) \implies (r.Loc = r'.Loc).$$

Here r and r' are two variables that bind to FactIDs in the fact table D. As another example, we can write the second constraint in Example 1 "`Madison` *can be either a person name or a city, but not both, and so is the case with* `Dells`*" as follows:*

$$(p_3.Loc = \texttt{Madison}) \Rightarrow (\neg exists(p_4))$$
$$(p_4.Loc = \texttt{Dells}) \Rightarrow (\neg exists(p_3)).$$

Note that here p_3 and p_4 are not variables, but refer to specific factIDs in the fact table D.

As yet another example, for the fact table in Figure 2.a we can write the constraint from Example 1, "if repairs p_4 and p_5 take place in Mad, then repair p_6 refers to a car of model $F150$ and repair p_3 does not exist":

$$(p_4.Loc = \texttt{Madison}) \wedge (p_5.Loc = \texttt{Madison}) \Rightarrow (p_6.Truck = \texttt{F150}) \wedge (\neg exists(p_3))$$

As these examples demonstrate, this constraint language is relatively simple, and yet already allows us to write expressive constraints. Our experience with fact extraction in two real-world domains – Web data in the DBLife system of the Cimple project [14] and emails in the Avatar project [23] – shows that we can use this language to capture many domain constraints in those domains. In Section 8 we briefly discuss generalizing this language to more expressive types of constraints (e.g., aggregations over a set of facts) as future work. We do not consider detecting inconsistent constraints, and refer the reader to [9] for details.

4.2 Semantics

Let D be a fact table and C be a set of constraints as defined earlier. We now describe what it means to answer OLAP queries over D, given C. Recall from Section 3 that in the case of no constraint, to answer an OLAP query Q, we

1. create an EDB table D', whose tuples are claims, from D, using an allocation policy,
2. select claims from D' (one for each fact) to generate multiple possible worlds $W_1, \ldots, W_m$, then compute a probability p_i for each world W_i, and
3. compute the expected answer over all these worlds: $\sum_{i=1}^{m} p_i * ans(Q, W_i)$ and return it as the answer $ans(Q)$ for Q.

To accommodate constraints C, we keep the above multiple-world semantics, but discard those that do not satisfy C. To do so, we start with the following notion:

DEFINITION 6 (VALID WORLD). *Let W be a world created by selecting claims, one for each fact, from an EDB table, as described earlier, and let C be a set of constraints. Then each constraint $c_i \in C$ can be evaluated to TRUE or FALSE on W. In particular, if c_i contains variables, then it evaluates to TRUE iff all possible bindings of the variables to factIDs in W make c_i evaluate to TRUE. We say W is valid (wrt C), or W satisfies C, iff all constraints in C evaluates to TRUE on W.*

EXAMPLE 5. *Consider a simple example with two facts r_1 = (WI,F150) and r_2 = (WI,F150), and the single constraint "two facts with the same model must have the same location". There are 4 possible worlds, since both r_1 and r_2 have two possible completion claims: {(Madison,F150),(Dells,F150)}. The only valid worlds are where the same claim is selected for both r_1 and r_2. For example, if (Madison,F150) is selected for r_1 in world W, then (Madison,F150) must be selected for r_2 for W to be valid.*

Suppose after discarding invalid worlds from $W_1, \ldots, W_m$, we obtain the valid worlds $W_{i1}, \ldots, W_{ik}$. Recall that they have been assigned probabilities $p_{i1}, \ldots, p_{ik}$, which are now incorrect because most likely these probabilities sum to less than 1. In the absence of any additional information, a common solution for revising these probabilities is to scale them proportionally, so that they sum to 1 [19]. We adopt this solution for our context. We now can define our query semantics as follows:

DEFINITION 7 (CONSTRAINT-BASED QUERY SEMANTICS). *Given a fact table D and a set of constraints C, let $W_{i1}, \ldots, W_{ik}$ be the valid worlds (wrt C) with revised probabilities $p_{i1}, \ldots, p_{ik}$, as described above. Then for any OLAP query Q, we return the expected answer over the valid worlds $\sum_{j=1}^{k} p_{ij} * ans(Q, W_{ij})$ as the answer for Q over D in the presence of C, denoted $Q(D, C)$.*

5. QUERY ANSWERING WITH MDB D*

We now describe our solution for answering OLAP queries over an imprecise database D, given a set of constraints C. We begin by defining the types of queries we consider. While the OLAP paradigm offers a rich array of query operators, the basic query consists of selecting a value from $dom(A_i)$ for each dimension i, and applying an aggregation operator to a particular measure attribute.

DEFINITION 8 (BASIC QUERY AND QUERY REGION [8]). *For each dimension i, define a query domain, denoted $qdom(A_i)$, to be some non-empty subset of $dom(A_i)$. A basic query Q over a fact table D with schema $\langle A_1, \ldots, A_k, M_1, \ldots, M_n \rangle$ has the form $Q(a_1, \ldots, a_k; M_j, \mathcal{A})$, where (a) each $a_i \in qdom(A_i)$ and together $a_1, \ldots, a_k$ describe the k-dimensional region being queried, denoted $reg(Q)$, (b) M_j is a measure of interest, and (c) $\mathcal{A}$ is an aggregation function.*

In this paper, as in [8], we consider the common aggregation functions Sum, Count, and Average. All general queries (e.g., *roll-up, slice, drill-down, pivot*, etc.) can be described in terms of repeated applications of basic queries. Hence, we focus on answering basic queries in the presence of constraints.

EXAMPLE 6. *Figure 3.a shows a database D of four facts and a query Q = "What is the Sum of Sales for Mad?" (shorthand for Madison) over D. For query Q, a_1 is Model, with value ALL (i.e., the one that contains all singleton values in $dom(A_1)$), a_2 is Loc, with value Mad, M_j is Sales, and $\mathcal{A}$ is Sum. Figure 3.b shows a multi-dimensional view of D. $reg(Q)$ is the dotted region in this view.*

Recall from Definition 7 that answering a (basic) query Q reduces to evaluating it over all valid worlds. This basic approach is clearly impractical. Hence we seek a more efficient solution. In the rest of this section we first define the notion of a marginal database MDB $D*$, then prove that we can answer Q efficiently using $D*$. Later in Section 6 we show how to efficiently construct $D*$.

DEFINITION 9 (MARGINAL DATABASE MDB $D*$). *Let D be an imprecise database and D' be an EDB obtained from D via an allocation policy. Let C be a set of constraints, and $\mathcal{W}$ be the set of all valid worlds (i.e., those that are derived from D' and satisfy C, see Section 4).*

Recall that each claim t in D' consists of a precise fact f_t and an allocation weight w_t. Let m_t be the total probability of all worlds in $\mathcal{W}$ where f_t is true. That is, $m_t = \sum_{W \in \mathcal{W}} p(W)$, where f_t is true in W and $p(W)$ is the probability of W. Then we refer to m_t as the marginal probability of f_t, and refer to the pair (f_t, m_t) as a marginal tuple. We refer to the set of all marginal tuples as the marginal database (MDB for short) $D*$.

EXAMPLE 7. *Continuing with Example 6, Figure 3.c shows an EDB D', obtained via an allocation policy, and a set C of just one constraint, "two facts with the same model must have the same location", over D. From D' and C we can compute the MDB $D*$ in Figure 3.d.*

It is important to note that each tuple in $D*$ has a corresponding tuple in D'. Furthermore, $D*$ depends only on D, a particular allocation policy, and a set of constraints C. It does not depend on Q. Hence, once constructed, $D*$ can be used to answer all queries.

Specifically, if Q is Sum, then we can compute $Q(D*)$ to be the weighted sum over all cells of $reg(Q)$. Formally, $Q(D*) = \sum_{f_t \in reg(Q)} Q(f_t) \cdot m_t$, where (f_t, m_t) ranges over all tuples in $D*$, and $Q(f_t)$ is the value of M_j, the measure of interest of Q (see Definition 8).

EXAMPLE 8. *The answer for query Q in Figure 3.a is 1*0.78 + 4*0.78 + 3*0.72 + 2*0.72 = 7.5*

We can compute $Q(D*)$ for Count and Average (see [8] for details) in an analogous fashion. Overall, $Q(D*)$ can be computed via a single scan over $D*$. We note that many existing optimizations for evaluating OLAP queries over a fact table (e.g., materialized views and indexes) could be used to further speed up query processing after $D*$ has been materialized.

We now prove that $Q(D*)$ is the same as $Q(D, C)$ in Definition 7. This result is important because it suggests that we can focus our effort on constructing MDB $D*$, which we do in Section 6. For space reasons, we will state and prove the result for Sum only, though the result and proof for Count and Average are similar.

PROPOSITION 1. *Let C be a set of constraints, and $Q(D, C)$ be the answer to Q over an imprecise database D (Definition 7). Let $D*$ be an MDB obtained from D and C. Suppose Q is a Sum query. Then $Q(D, C) = Q(D*)$.*

PROOF. From Definition 7, we have

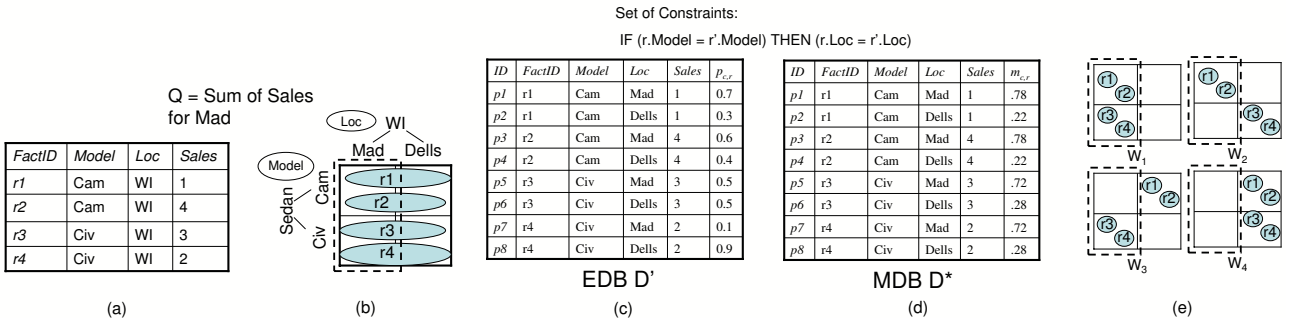
$$Q(D, C) = \sum_{i:W_i \text{ valid}} p_i * Q(W_i) \quad (1)$$

Let the $D_j, j = 1, \ldots, k$ be an arbitrary k-partitioning of the facts. We refer to the contents of the partition for world W_i as $D_{j,i}$. Then, from the distributivity of Sum, we obtain

$$Q(D, C) = \sum_{i:W_i \text{ valid}} p_i * (\sum_{j} Q(D_{j,i})) = \sum_{j_1^k} \sum_{i:W_i \text{ valid}} (p_i * Q(D_{j,i})) \quad (2)$$

where $Q(D_{j,i})$ is the result for Q on the facts in partition $D_{j,i}$ for world W_i. Let $Y_{i,r,Q}$ be a variable that takes value 1 if fact r completes to a cell $c \in reg(Q)$ in the valid world W_i, and 0 otherwise. Let v_r be the measure value for fact r. Then we have

$$Q(D, C) = \sum_{j_1^k} \sum_{i:W_i \text{ valid}} p_i * (\sum_{r \in D_{j,i}} v_r * Y_{i,r,Q}) \quad (3)$$



FactID	Model	Loc	Sales
r1	Cam	WI	1
r2	Cam	WI	4
r3	Civ	WI	3
r4	Civ	WI	2

ID	FactID	Model	Loc	Sales	$p_{c,r}$
p1	r1	Cam	Mad	1	0.7
p2	r1	Cam	Dells	1	0.3
p3	r2	Cam	Mad	4	0.6
p4	r2	Cam	Dells	4	0.4
p5	r3	Civ	Mad	3	0.5
p6	r3	Civ	Dells	3	0.5
p7	r4	Civ	Mad	2	0.1
p8	r4	Civ	Dells	2	0.9

ID	FactID	Model	Loc	Sales	$m_{c,r}$
p1	r1	Cam	Mad	1	.78
p2	r1	Cam	Dells	1	.22
p3	r2	Cam	Mad	4	.78
p4	r2	Cam	Dells	4	.22
p5	r3	Civ	Mad	3	.72
p6	r3	Civ	Dells	3	.28
p7	r4	Civ	Mad	2	.72
p8	r4	Civ	Dells	2	.28

Figure 3: An example to illustrate query answering with MDB $D*$.

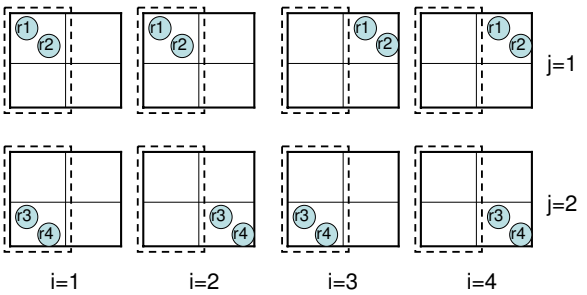


Figure 4: Visualization of Proof Example

By pulling v_r out, we obtain

$$Q(D,C) = \sum_{j_1^k} \sum_{r \in D_{j,i}} v_r * (\sum_{i:W_i \text{ valid}} p_i * Y_{i,r,Q}) \quad (4)$$

$$= \sum_{r \in D} v_r * (\sum_{c:c \in reg(Q)} m_{c,r}) = Q(D*). \quad (5)$$

□

EXAMPLE 9. *Continuing with Example 7, given the single constraint in Figure 3, only four out of 16 possible worlds satisfy the constraint. Figure 3.e. shows these four worlds. Here each dotted box denotes the query region $reg(Q)$. Recall that the probabilities of the valid worlds are computed by normalizing their "old" probabilities. For example, the "old" probability p_1 of world W_1 is (0.7 * 0.6 * 0.5 * 0.1) = 0.021 Similarly, p_2 = 0.189, p_3 = 0.006, and p_4 = 0.054.*

So the revised p_1, denoted p_1^N, is $p_1^N = \frac{p_1}{p_1+p_2+p_3+p_4} = \frac{0.021}{0.021+0.189+0.006+0.054} = \frac{0.021}{0.27} = 0.08$. Similarly, p_2^N = 0.7, p_3^N = 0.02, p_4^N = 0.2.

W.l.o.g., assume we create two partitions $D_1 = \{r_1, r_2\}$ and $D_2 = \{r_3, r_4\}$, thus, i = 4 worlds, and j = 2 partitions for the example. To visualize the summations in the proof, consider Figure 4. The summation in Equation 2 of the proof can be thought of as evaluating the Sum query separately over each of these 8 "datasets". In Equation 2, we process all partitions for each world together (i.e. process these "datasets" column-wise). In Equation 3, we now process the same partition in each possible world together (i.e., process these datasets row-wise). Now, we describe how each partition is processed (Lines 4 and 5). The value for Q on partition D_i can be considered the Sum of the Sales *measure for facts which complete to a cell $c \in reg(Q)$. For example, consider fact r_1. The only cell r_1 completes to inside $reg(Q)$ is (Cam, Mad). Thus, $Y_{r_1,i,Q}$ = 1 for i= 1,2 (i.e., worlds W_1 and W_2) with normalized weights* 0.08 *and* 0.7 *respectively; and $Y_{r_1,i,Q}$ = 0 for $i = 3, 4$. Thus, fact r_1 contributes to the answer for Q in 0.78 of the possible worlds, (i.e., $\sum_i p_i^N * Y_{r,i,Q}$ = 0.08(1) + 0.7(1) + 0.02(0) + 0.2(0) = 0.78.*

This is the sum of the $m_{c,r}$ values for the claims in MDB $D*$ for fact r_1 where r_1 completes to a cell $c \in reg(Q)$. This can also be interpreted as the expected contribution of r_1 to the answer to Q is $0.78 * sales(r_1) = 0.78 * 1 = 0.78$, which is the $m_{c,r}$ for r_1 in $D*$.

6. COMPUTING DATABASE MDB D*

We have shown that OLAP queries can be answered quickly, given a MDB $D*$. We now show how to construct MDB $D*$ from an EDB D' and a set of constraints C. Definition 9 immediately suggests a naive algorithm to compute $D*$: enumerate all possible worlds W_i (i.e., by selecting one claim per fact from D'), retain only those that are valid (with respect to C), then compute the marginal $m_{c,r}$ for each claim tuple c as the probability portion of valid worlds where claim c is selected for r.

This algorithm is clearly infeasible in practice, due to the exponential number of possible worlds (in the size of D). Let $|D|$ be the size of D, and $|c|$ be the maximal number of claims in D' for any fact. Then, the above has complexity $O(|c|^{|D|})$.

To address this problem, our solution is to (a) exploit the regularities in the constraint space to decompose D' into independent *connected components*, (b) exploit the regularities in the fact space to process each component in isolation, yielding a portion of the MDB $D*$, then (c) combining these portions to obtain the entire MDB $D*$. To further speed up these steps, we employ a RDBMS whenever possible. The rest of this section elaborates on the above steps.

6.1 Decomposing D' into Components

We first introduce the notion of constraint hypergraph, which we use to capture the regularities in the constraint space. We then show how to use this hypergraph to decompose D' into connected components.

6.1.1 Constraint Hypergraph

We begin by establishing several notions.

DEFINITION 10 (CONFIGURATION). *Let $S = \{r_1, \ldots, r_j\}$ be a subset of j facts in D. Let $claims(r)$ be the set of possible claims for r in D'. We refer to an element $(c_1, \ldots, c_j) \in claims(r_1) \times \cdots \times claims(r_j)$ as* configuration *C_S for $\{r_1, \ldots, r_j\}$.*

DEFINITION 11 (VALID CONFIGURATION). *Configuration $C_S = (c_1, \ldots, c_j)$ for fact set $S = \{r_1, \ldots, r_j\}$* violates *a constraint c if there exists a subset of C_S violating the conjunction of atoms in c. Otherwise, configuration C_S* satisfies *constraint c.*

C_S is a valid configuration *if all constraints in constraint set C are satisfied; otherwise, C_S is* invalid*. A configuration for all facts in D is a* possible world*, and a valid configuration for D is a* valid possible world*.*

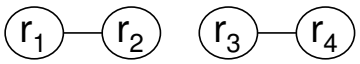


Figure 5: Hypergraph for the example in Figure 3

By this definition, a configuration C_S may implicitly satisfy a constraint c which is not directly applicable to C_S (e.g., c has more than j variables or c mentions factIDs for facts not in S).

DEFINITION 12 (CONSTRAINT HYPERGRAPH). *We define the* constraint hypergraph $G = (V, H)$ *as follows: For each fact* $r \in D$, *create a corresponding node* $v_r \in V$. *Let* $S = \{r_1, \ldots, r_k\} \in D$ *be a set of facts such that (a) some configuration* $(c_1, \ldots, c_k)$ *of* S *violate a constraint in* C *and (b) there exists no subset of* S *where a configuration of that subset violates a constraint in* C. *For each such* S, *introduce an undirected hyperedge* $(v_{r_1}, \ldots, v_{r_k}) \in H$.

EXAMPLE 10. *Consider fact table* D *in Figure 3.a, and a set* C *of just one constraint "two facts with the same model must have the same location". The resulting hypergraph* $G = (V, H)$ *for* D, *shown in Figure 5, has a node in* V *for each of the 4 facts in* D. H *has two hyperedges* $\{r_1, r_2\}$, $\{r_3, r_4\}$. *For example, the edge* $\{r_1, r_2\}$ *is added to* H *because the possible configuration [* `(Cam, Mad), (Cam, Dells)` *] for* r_1 *and* r_2 *violates the constraint. Likewise, the edge* $\{r_3, r_4\}$ *is added since configuration [* `(Civ, Mad), (Civ, Dells)` *] for* r_3 *and* r_4 *violates the constraint.*

Observe that a node is added to the constraint graph for each *fact* in D, not each claim tuple in the EDB D'. We now describe how to use the constraint graph G to partition D' into sets of claim tuples which can be processed separately to assign the marginals.

DEFINITION 13 (INDEPENDENT/DEPENDENT FACTS). *Consider facts* $r, r' \in D$, *with possible claims* $claims(r)$, $claims(r')$ *respectively. Let* $\mathcal{W}_{c',r'}$ *be the set of valid possible worlds where claim* c' *is selected for fact* r'. *We define* $frac(c, r)$ *as the probability portion of all valid possible worlds where claim* c *is selected for* r, *and* $frac(c, r|\mathcal{W}_{c',r'})$ *as the probability portion of* $\mathcal{W}_{c',r'}$ *where claim* c *is selected for fact* r.

If $frac(c, r) = frac(c, r|\mathcal{W}_{c',r'})$ *for all* $c' \in claims(r')$ *and* $c \in claims(r)$, *we refer to* r *and* r' *as* independent. *Otherwise,* r *and* r' *are* dependent.

EXAMPLE 11. *Continuing with Example 10, for the given constraint in Figure 3 the four valid possible worlds are shown in Figure 3.e, with each valid world having the following "revised" probabilities, respectively:* p_1^N *= 0.08,* p_2^N *= 0.7,* p_3^N *= 0.02,* p_4^N *= 0.2. (see Example 9).*

Consider facts r_1 *and* r_2. *The weighted fraction of valid worlds where claim* c_1 *= (Cam,Mad) is selected for* r_1 *is 0.78, (i.e.* $frac(c_1, r_1)$ *= 0.78), since claim (Cam,Mad) is selected for* $r1$ *in worlds* w_1 *and* w_2, *and* $p_1^N + p_2^N$ *= 0.08 + 0.7 = 0.78. Fact* r_2 *has 2 possible claims, (Cam,Mad) and (Cam,Dells), with (Cam,Dells) selected for* r_2 *in worlds* w_3 *and* w_4. *However, in neither* w_3 *nor* w_4 *is claim* $c1$ *selected for* r_1 *(i.e.,* $frac(c_1, r_1|\mathcal{W}_{(Cam,Dells),r_2}) = 0$*). Since* $frac(c_1, r_1|\mathcal{W}_{(Cam,Dells),r_2}) \neq frac(c_1, r_1)$, r_1 *and* r_2 *are dependent.*

In contrast, consider facts r_1 *and* r_3, *where* r_3 *has two possible claims,* `(Civ, Mad)` *and* `(Civ, Dells)`. *It is easily verified from Figure 3.e that* $frac(c_1, r_1) = frac(c_1, r_1|\mathcal{W}_{(Civ,Mad),r_3})$ *=* $frac(c_1, r_1|\mathcal{W}_{(Civ,Dells),r_3})$ *= 0.78, and that a similar result holds for the other possible claims of fact* r_1. *Thus, facts* r_1 *and* r_3 *are independent.*

Intuitively, claims in MDB $D*$ for set of facts S can have marginals assigned separately of other facts in $D - S$ if each fact in S is independent of all facts in $D - S$. The next theorem gives the necessary condition (in terms of G) for two facts r, r' to be dependent.

THEOREM 1. *Consider facts* $r, r' \in D$. *The existence of a path between* r, r' *in constraint hypergraph* G *is a necessary condition for* r, r' *to be dependent.*

The existence of a path in the constraint hypergraph between r, r' is not a sufficient condition for r, r' to be dependent; see details in [9].

COROLLARY 1. *If there is no path in constraint hypergraph* G *between* r, r', *then* r, r' *are independent.*

6.1.2 Generating Connected Components

Although identifying connected components in a given disk resident graph is a well-studied problem, e.g., [2, 13, 31], there exists no straightforward application of these solutions to our problem setting, since we consider hypergraphs and the constraint hypergraph is not provided as input (we only have D').

The algorithm we propose, called *GenerateComponent*, takes as input a given EDB D' (as a RDBMS table with the schema given in Definition 3) and constraints set C, and outputs the hyperedges in G. The algorithm generates the hypergraph G in a "component-wise" fashion, generating all edges for each connected component G_i together. Thus, the connected components are identified during the hypergraph generation process.

GenerateComponent starts by creating a relational database table CompID with schema (factid, cid), which stores the component identifier cid assigned to fact $factid$. Initially, all facts are given a special "unassigned" cid. The algorithm continually selects a node with unassigned cid as a "seed node" v_r, and performs a breath-first enumeration of G from v_r until the component containing v_r is completely enumerated. The edges in the component are generated separately for each constraint $c_i \in C$ and stored in a separate database table $CiEdges$ (e.g., edges for constraint c_1 are stored in table $C1Edges$, edges for c_2 in $C2Edges$, etc.). The schema for each $CiEdge$ table is $(fid_1, \ldots, fid_k)$ for k-constraint c_i. At each step, the algorithm identifies a set of border nodes which require expansion, and these are stored in the table activeSet. Edges are generated only from nodes in activeSet by executing for each constraint c of the form $A \Rightarrow B$ a SQL query of the form:

`SELECT` D1.factid, D2.factid, . . . , Dk.factid
`FROM` $activeSet$ AS D1, EDB AS D2, . . . , EDB AS Dk
`WHERE` [logic for A] AND [¬ logic for B] AND [D2.fid < . . . < Dk.fid]

EXAMPLE 12. *The constraint from Example 11 "all facts with the same location must have the same model" generates the SQL query:*

`SELECT` *D1.factid, D2.factid*
`FROM` *activeSet AS D1, EDB AS D2, EDB AS D3*
`WHERE` *D2.loc = D3.loc* `AND` *D2.model != D3.model* `AND` *D2.fid < D3.fid*

After all of the queries generating edges for each constraint have been executed, activeSet is set to nodes with unassigned cid in the edges created by these queries, and the cid for the corresponding tuples in CompID are updated to the current component id. This query is executed until activeSet becomes empty. At this point, all edges in the component have been generated. The algorithm repeats this process for a new "seed node"; if none is available, the algorithm terminates. The complete pseudocode for GenerateComponent is listed in Algorithm 1.

THEOREM 2 (CORRECTNESS). *The GenerateComponent Algorithm correctly identifies all connected components in the constraint hypergraph* G.

Algorithm 1 GenerateComponent Algorithm

```
1: // initialize CompID; CompID stores the component assignment
2: CREATE TABLE CompID(factid, cid) AS SELECT DISTINCT factid, -1 FROM EDB
3: CREATE TABLE activeSet(fid);
4: initialize current component id ccid to 0;
5: // while facts are not assigned to connected components
6: while ( 0 < SELECT COUNT (*) FROM CompID C WHERE C.cid == -1) do
7:   // increment the current connected component id
8:   ccid ← ccid + 1;
9:   select fact r not yet assigned to a component (i.e., tuple in compID s.t. compid.cid = -1)
10:  initialize activeSet to r; UPDATE compID SET cid = ccid WHERE factid = r.factID
11:  while (activeSet has tuples) do
12:    // generates table with edges for each constraint c_i ∈ C
13:    for (each constraint c_i ∈ C of form A ⇒ B) do
14:      // materialize conflict edges from activeset for c_i
15:      INSERT INTO TABLE CiEdges (fid1,fid2,. . . ,fidk)
16:      SELECT D1.factid, D2.factid, . . . , Dk.factid
17:      FROM activeSet AS D1, EDB AS D2, . . . , EDB AS Dk
18:      WHERE [logic for A] AND [¬ logic for B] AND [D1.fid < D2.fid < . . . < Dk.fid]
19:    // find the set of active nodes
20:    CREATE TABLE activeTable AS
21:    SELECT fid1 FROM C1Edges,CompID C WHERE (C.factid = fid1) AND (C.cid == -1) UNION . . .
22:    SELECT fidk FROM CkEdges,CompID C WHERE (C.factid = fidk) AND (C.cid == -1)
23:    // update the component ids for the activeSet
24:    UPDATE CompID SET cid = ccid
25:    WHERE CompID.factid IN (SELECT * from activeTable)
26:    update activeSet to activeTable
```

6.2 Processing Components to Create Portions of MDB $D*$

Each connected component G_i of constraint hypergraph G corresponds to a partition D'_i of EDB D' such that claims in D'_i can be assigned marginals by only processing other claims in D'_i. This section describes the process to assign the marginal $t.m_{c,r}$ to each claim in each identified component, thus generating the claims in MDB $D*$ corresponding to D_i (see Figure 1.c).

The results in Section 6.1.1 suggest the following *Component-wise Naive Algorithm*: For each connected component $G_i = (V_i, H_i)$ in G, enumerate every possible configuration for facts in imprecise database D corresponding to V_i. Then, for each claim t in MDB $D*$ for a fact in V_i, assign $t.m_{c,r}$ the weighted portion of valid configurations where claim c is selected for r. The complete MDB $D*$ is obtained by concatenating the portion output for each component.

The complexity of this Component-wise Naive Algorithm is $O(N * |c|^m)$, where N is the number of components, $|c|$ is the maximal number of claims in D' for any fact, and m the maximal number of imprecise facts in any component. Although this compares favorably with the complexity for the brute force algorithm, which was $O(|c|^{|D|})$, we will present a more efficient algorithm than Component-wise Naive in this section.

6.2.1 Reducing the Number of Enumerated Configurations

We now present a more efficient algorithm for assigning marginals, called *ProcessComponent*, which requires enumerating fewer configurations that Component-wise Naive. Given the subgraph for $G_i = (V_i, H_i)$ and the corresponding partition of EDB D'_i as input, ProcessComponent assigns a marginal $t.m_{c,r}$ to each corresponding claim t in $D*$. The final Marginal Database $D*$ is obtained by concatenating the outputs of ProcessComponent for each G_i together.

At the highest level, ProcessComponent proceeds as follows: For every claim t in D'_i, we maintain a running sum of configuration weights for valid configurations C_{V_i} where claim c is selected for fact r. The nodes in V_i are partitioned into two sets I, J. Since the nodes in V_i correspond to facts in the imprecise database D, we refer to V_i as nodes or facts interchangeably. We enumerate each possible configuration C_J for facts in J, and compute *configWeight* as the total weight of all valid configurations of $V_i = I \cup J$ for which C_J is a sub-configuration for J (i.e., the claims selected for facts in $J \in V_i$ are given by C_J).

For each C_J, *configWeight* is computed as follows: For each fact $r_I \in I$, we find the set of claims $c \in claims(r_I)_{|C_J}$ for r_I such that (c, C_J) is a valid configuration for $r_I \cup J$. Let the sum of allocations for these claims be $sumValid(r_I)_{|C_J}$. *configWeight* is then given by $(\prod_{r_I \in J} p_{c,r_I}) * (\prod_{r_I \in I} sumValid(r_I)_{|C_J})$, and is added to the running sum for each claim tuple corresponding to a completion in C_{V_i}. The complete pseudocode for ProcessComponent is listed in Algorithm 2.

PROPOSITION 2 (COMPLEXITY). *Let N be the number of components, $|c|$ the maximal number of claims in D' for any fact, and m be the maximal number of imprecise facts in any component. Assume we partition these m imprecise facts into two sets I, J of size m_I, m_J s.t. m_I is the size of set I, m_J is the size of set J, and $m = m_I + m_J$. Then, the complexity of ProcessComponents is $O(N * |c| * m_I * |c|^{m_J})$*

We observe that the complexity for ProcessComponents is an improvement over Component-Naive, since ProcessComponent is guaranteed to have a smaller exponent than Component-Naive (i.e., $m_J < m$). For the practical-sized datasets with up to several million facts considered in our experiments (Section 7.1), the complexity of ProcessComponent was not an issue. How far this extends to other practical datasets is an intriguing issue for future work. The reason for this was that the observed value for m_J tended to be small (i.e., less than 20 for datasets we consider). We note the Naive Algorithm with complexity $O(|c|^{|D|})$ is clearly intractable for practical-sized datasets. We now provide intuition for selecting the "best" set J of imprecise facts in the component.

Algorithm 2 ProcessComponent Algorithm

```
1: Input: EDB partition D'_i, Constraint Hypergraph Component G_i
2: Output: Marginals for MDB D*_i (corresponds to D'_i)
3: I ← BestNonAdjecentSet(G_i)
4: J ← V_i − I
5: create MDB entries D*_i for D'_i
6: initialize array sumValid[] // stores Σ_I p_{c,r_I} where c, C_J valid
7: for (each valid configuration C_J of J) do
8:   for (each fact r_I ∈ I) do
9:     sumValid[r_I] ← 0
10:    for (each c ∈ comp(r)) do
11:      if (c ∪ C_J is valid configuration of {r_I ∪ J}) then
12:        sumValid[r_I] ← sumValid[r_I] + p_{c,r}
13:  // p_{c,r_J} is allocation for completion used in C_J for r_J
14:  configWeight ← (∏_J p_{c,r_J}) * (∏_{r_I ∈ I} sumValid[r_I])
15:  totalWeight ← totalweight + configWeight
16:  // update weights for each fact in J
17:  for (each fact r_J ∈ J) do
18:    t.m_{c,r} ← t.m_{c,r} + configWeight
19:  // update weights for facts in I
20:  for (each fact r_I ∈ I) do
21:    for (each c ∈ comp(r)) do
22:      if (c ∪ C_J is valid configuration of {r_I ∪ J}) then
23:        t.m_{c,r} ← p_{c,r} * configWeight
24: // normalize the weights in t.m_{c,r}
25: for (each t ∈ M_i) do
26:   t.m_{c,r} ← t.m_{c,r} / totalWeight
```

DEFINITION 14 (NON-ADJACENT SET). *Consider a connected component $G_i = (V_i, H_i)$ in G. We refer to $I \subseteq V_i$ as a* non-

adjacent set *if there does not exist a pair of nodes $v, v' \in I$, such that v, v' share an edge in H_i.*

A non-adjacent set is equivalent to the notion of *strong independent set* in a hypergraph $G = (V, H)$, which is defined as a set of nodes $I \subseteq V$ such that no pair $v, v' \in I$ share a hyperedge in H [18].

THEOREM 3. *Consider connected component $G_i = (V_i, H_i)$ with corresponding partition D'_i of EDB D' (i.e., the claims in D' for facts corresponding to nodes in V_i). Let $V_i = I \cup J$. If I is a non-adjacent set in G_i, then ProcessComponent correctly assigns marginals to claim tuples in D'_i.*

6.2.2 Identifying Non-adjacent Sets

The result in Theorem 3 holds for any possible non-adjacent set in V_i. We now propose a cost model for comparing the cost of marginal assignment by using the various possible non-adjacent sets in G_i.

DEFINITION 15 (COST MODEL). *Let $size(v_r)$ be the number of completions for fact r. Let α be a constant capturing the cost of enumerating a configuration. Likewise, β is a constant for processing a single completion for a single fact. Assume $V_i = I \cup J$, where I is a non-adjacent set. The cost of processing component G_i using sets I and J is given by*

$$cost(G_i, I, J) = \alpha(\prod_{v_J \in J} size(v_J)) * max\{1, (\beta(\sum_{v_I \in I} size(v_I))\}$$

The second term is required to be at least 1, to handle the special case when $I = \emptyset$.

In practice $\alpha > \beta$, since enumerating a configuration involves more work than processing facts separately. For this case, Component-Wise Naive Algorithm has the highest possible cost. The lowest possible cost would be obtained by making the product of the size(v_I) for $v_I \in I$ as large as possible.

PROPOSITION 3 (OPTIMAL). *1) For $\alpha > \beta$, $cost(G_i, I, J)$ in our model is optimized when I is assigned the non-adjacent set with the largest product of sizes (i.e., I s.t. $\prod_{v_I \in I} size(v_I)$ is maximized). 2) The problem BEST-NON-ADJACENT of finding the I which minimizes cost(G_i,I,J) is an NP-complete problem.*

We can trivially reduce the problem of finding the maximal weighted strong independent set in an undirected hypergraph, which is NP-complete [18], to BEST-NON-ADJACENT. The hypergraph given as input to the weighted strong independent set problem is given as input to BEST-NON-ADJACENT , along with $\alpha = |V_i|, \beta = 1$, where $|V_i|$ is the size of V_i. The non-adjacent set returned for I is the maximal weight strong independent set in G. Thus, a polynomial time algorithm cannot exist for BEST-NON-ADJACENT. [18] presents negative theoretical results on the existence of good approximation algorithms for the weighted strong independent set problem and the related problem of hypergraph coloring.

We now present an algorithm *MaxNonAdjacent* which find a maximal non-adjacent set in a given hypergraph component $G_i = (V_i, H_i)$, which is an approximation of the maximal non-adjacent set optimizing our cost model. Our algorithm is semi-external, since we are only required to store in memory at all times a bit-vector $status$ with an entry $status(v)$ for each node $v \in V_i$ indicating whether v is in the maximal non-adjacent set or not. Essentially, the MaxNonAdjacent algorithm scans the set of hyperedges H, and greedily maintains the "best" strong independent set of V for the edges seen. The pseudocode is listed in Algorithm 3.

Algorithm 3 MaxNonAdjacent Algorithm

```
1: Input: Hypergraph G_i = (V_i, H_i)
2: Output: Returns approximate maximal weighted non-adjacent set I ⊆ V
3: allocate bit-vector status with a bit for each v ∈ V_i
4: // define NON-ADJACENT = 0, ADJACENT = 1
5: for (each v_i ∈ V) do
6:    Initialize status(v) to NON-ADJACENT
7: for (each edge h ∈ H) do
8:    select node v ∈ h with highest weight s.t. status(v) = NON-ADJACENT.
9:    update status(v') = ADJACENT for all other nodes v' ∈ h
10: I ← subset of nodes in V with status NON-ADJACENT
11: return I
```

6.3 Combining $D*$ Portions to Obtain Complete $D*$

We use GenerateComponent to identify the portion of G for each connected component G_i=(V_i, H_i). After generating the component edges, ProcessComponent is called to generate MDB $D*$ tuples for facts corresponding to nodes in V_i. Intuitively, identifying strongly connected components in constraint graph G is equivalent to partitioning EDB D' into sets of claim tuples such that marginals can be assigned to claims in each set independently of the other sets. The following formalizes this intuititon.

THEOREM 4. *Let $G_i = (V_i, H_i)$ be a connected component in constraint graph G, and let D'_i be the EDB claims for the facts in D which correspond to nodes in V_i (see Definition 12). Then, the marginals for claims in D'_i can be assigned independently of claims not in D'_i using the ProcessComponent Algorithm.*

After ProcessComponent completes, the edges H_i may be discarded. After processing all components, all MDB $D*$ claim tuples have been generated. During algorithm execution, if ProcessComponent finishes and any MDB claim for a component is assigned a marginal of 0, then the constraint set is inconsistent. We can stop processing immediately, and marginals of 0 are assigned for all MDB entries in the component. When combining the portions of $D*$ together, we handle an inconsistent set of constraints by insuring all MDB claims have marginals set to 0, which is correct behavior by Definition 9.

7. EXPERIMENTAL RESULTS

To empirically evaluate the performance of the proposed algorithms, we conducted experiments using both real and synthetic data. The experiments were carried out on a machine running CentOS 4 with a dual Pentium 2.66 GHz processor, 2GB of RAM, and a single IDE disk. All algorithms were implemented as Java applications that accessed a local instance of IBM DB2 UDB Version 8.1 using JDBC to interface with the database.

Since existing data warehouses cannot directly support multi-dimensional imprecise data, obtaining "real-world" datasets is difficult. However, we were able to obtain one such real-world dataset from an anonymous automotive manufacturer. The fact table contains 797,570 facts, of which 557,255 facts were precise and 240,315 were imprecise (i.e., 30% of the total facts are imprecise). There were 4 dimensions, and the characteristics of each dimension are listed in Table 3. Two of the dimensions (*Sr-Area* and *Brand*) have 3 level attributes (including `ALL`), while the other two (*Time* and *Location*) have 4.

Each column of Table 3 lists the characteristics of each level attribute for the particular dimension, and ordered from top to bottom in decreasing granularity. Thus, the bottom attribute is the cell-level attribute for the dimension. The two numbers next to each attribute name are, respectively, the number of distinct values the attribute can take and the percentage of facts that take a value from that attribute for the particular dimension. For example, for the *Sr-Area*

dimension, 92% of the facts take a value from leaf-level *Sub-Area* attribute, while 8% take a value from the *Area* attribute.

Sr-Area	*Brand*	*Time*	*Location*
ALL(1)(0%)	ALL (1)(0%)	ALL (1)(0%)	ALL (1)(0%)
Area(30)(8%)	Make(14)(16%)	Quarter(5)(3%)	Region(10)(4%)
Sub-Area(694)(92%)	Model(203)(84%)	Month(15)(9%)	State(51)(21%)
		Week(59)(88%)	City(900)(75%)

Table 3: Dimensions of real dataset

Of the imprecise facts, approximately 67% were imprecise in a single dimension (160,530 facts), 33% imprecise in 2 dimensions (79,544 facts), 0.01% imprecise in 3 dimensions (241 facts), and none were imprecise in all 4 dimensions. For this dataset, no imprecise fact had the attribute value `ALL` for any dimension.

For several experiments synthetic data was generated using the same dimension tables as the real-world data. The process for generating synthetic data was to create a fact table with a specific number of precise and imprecise facts by randomly selecting dimension attribute values from these 4 dimensions. Each tuple was 40 bytes in size.

7.1 Algorithm Performance

We first evaluate the scalability of the Marginalization Algorithm along two dimensions. The first is the "complexity" of constraint set C and the second is the database size. While an obvious metric exists for the latter, defining an appropriate metric for comparing the "complexity" of different constraint sets on the same dataset is more involved. For example, an obvious metric like the number of constraints in C is potentially misleading since the amount of work required to evaluate a constraint c depends on how many facts c potentially applies to. For example, evaluating a single constraint "All facts with the same model have the same cost" involves more work than the constraint "All facts in the city Madison with the same model have the same cost," since the latter only applies to facts with location Madison and only pairs of these facts must be considered to evaluate the constraint.

Using this intuition, the metric we define to measure the complexity of a constraint is the number of potential *bindings* the constraint has in the fact table. In other words, the number of potential bindings for constraint c of the form $A \implies B$ is the number of times the conjunct of atoms in A could potentially hold in the fact table. The potential bindings for a constraint set C is the sum of potential bindings for each constraint $c \in C$.

For these experiments, we evaluate the scalability of the Marginalization Algorithm with respect to constraint complexity on the following three datasets: 1) the real Automotive dataset, 2) a randomly selected subset of 200,000 facts from Automotive (selected such that the 60,000 facts (30%) were imprecise), and 3) a synthetically generated dataset with 3.2 million total facts (of which 960,000 (30%) were imprecise). For each imprecise database D, we used Count-based allocation [8] as an off-line process to create the EDB D', which was stored in a relational table in the database. We randomly generated 35 constraint sets with varying degrees of complexity. Each constraint set was generated as follows: 1) First, we create a constraint "template" $A \implies B$ by: a) Randomly selecting 2 of the 4 dimensions,D_1, D_2 and create for A the following conjunct of atoms: "r.D_1 = r'.D_1 AND r.D_2 = r'.D_2", where r and r' are variables, and b) randomly selecting one of the remaining dimensions D_3 to create an atom for B with similar form. 2) We generate a set of constraints by repeatedly "instantiating" this template k times (for various k between 20 and 100) as follows: Randomly select values v_1, v_2 from $dom(D_1), dom(D_2)$ respectively, and add to the constraint the condition "where $D_1 = v_1$ AND $D_2 = v_2$." The result of each such "instantiation" is a constraint, and there are k constraints in the final constraint set.

All reported running times are "wall-clock" times and are warm numbers. Although the step to generate edges begins processing a constraint collection cold, the buffer pool is not flushed between the SQL queries over the EDB table used to generate hyperedges in the constraint hypergraph (see Section 6.1.2 for details). Thus, subsequent queries would be warm, since EDB D' tuples would still be in the buffer. The database was not tuned in any manner and no indexes were created over any tables. The buffer pool was set to 100 MB for all experiments.

The results for the three datasets are shown in Figures 6, with the figures displaying the running times versus number of possible bindings for each of the 35 randomly generated constraint sets, along with the most appropriate "best-fit" curves. Along with the total time, we also include the decomposition of the running times for the two main algorithm components, GenerateComponent and ProcessComponent (see Section 6). Separate best-fit curves were determined for each algorithm component, and indicate that while the running time of GenerateComponent grows linearly, the running time for ProcessComponent appears exponential. The main take-away from these results is that although our Marginalization algorithm is theoretically exponential (see complexity analysis in Section 6), it is indeed practical for real-world databases with millions of facts. Also, since Marginalization will be performed as an off-line process in most settings, this performance is reasonable.

We conducted a second set of experiments to explore the scalability of the Marginalization algorithm with respect to fact table size. For these experiments, we used the process described above to randomly generate a constraint set, and ran Marginalization on the same three datasets. The results are shown in Figure 7, with each figure showing for these datasets the total running time for Marginalization for a randomly selected constraint set. We omit results for other constraint sets since they were similar. From the graphs, we see the running time increases linearly as the dataset size increases. This indicates the scalability of the Marginalization Algorithm with respect to fact table size.

7.2 Component Size

The next set of experiments examines the size and number of connected components in the constraint hypergraph. For these experiments, we used the same three datasets and 35 randomly generated constraint sets used for experiments in Section 7.1. The results in Table 4 give the most extreme value observed over the 35 constraint sets, with each row corresponding to a dataset. The *Dataset* column contains the number of facts in the dataset. *Min # CC* and *Min # CC* contain respectively the minimum and maximum number of components observed in the constraint hypergraph over the 35 constraint sets. *Largest Comp* and *# Imp* contain respectively the total number of facts and the number of imprecise facts in the largest observed component. We note that the largest observed component also contained the most imprecise facts for all three datasets. Finally, *# Confs* gives the number of configurations enumerated to process this largest component. For each dataset, the largest component required enumerating the most configurations.

From Table 4 we see that for the datasets we consider no large connected components emerge in the constraint graph. The complexity analysis of the ProcessComponent step in Section 6.2 shows the number of configurations enumerated by ProcessComponent grows exponentially with respect to the number of imprecise facts in a component. Since no large connected component emerges, we see that the number of configurations enumerated for even the largest component remains reasonable despite the negative complexity results.

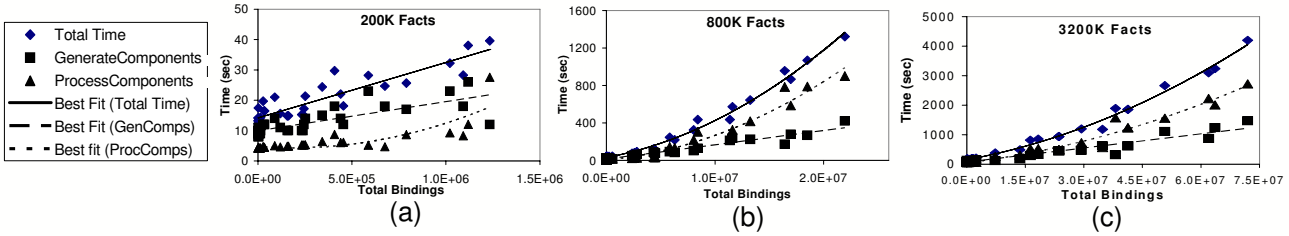


Figure 6: Scalability with respect to constraint complexity

Dataset	*Min # CC*	*Max # CC*	*Largest Comp*	*# Imp*	*# Confs*
200K	174198	205455	25 facts	7	2304
800K	691689	821733	78 facts	15	110592
3200K	2829814	3199461	120 facts	20	5308416

Table 4: Results for component size experiment

While these results demonstrate no large connected component emerges in the constraint hypergraph, they do not provide insight into the distribution of component sizes. We do not have enough information to conclude whether most of the components are extremely small (i.e, contain 1 or 2 imprecise facts) or if a significant number are "modestly" sized (i.e., contain 10 - 20 imprecise facts). This difference is significant since the number of configurations ProcessComponent enumerates to process a component grows exponentially with the number of imprecise facts it contains.

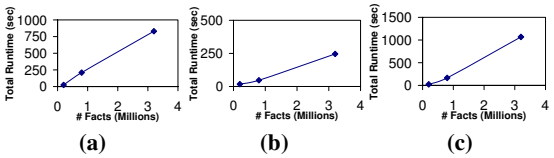


Figure 7: Scalability with respect to database size

The next set of results we present addresses this by reporting the distribution for imprecise fact count over the components for each dataset we consider. The results for each dataset over the 35 constraint sets used in the prior experiments are given in Figures 8.a-c. Each curve in a graph gives the number of components with the indicated number of imprecise facts as the constraint set size varies. E.g., the curve labelled "2 – 5" indicates the number of components with between 2 and 5 imprecise facts. Constraint set size k was defined in Section 7.1. We omit the curves for the range 0 to 1 imprecise facts since the number of configurations generated by these components is negligible (i.e., less that 5% of the total enumerated configurations) for the datasets we consider, and these negatively impact the readability of the graph.

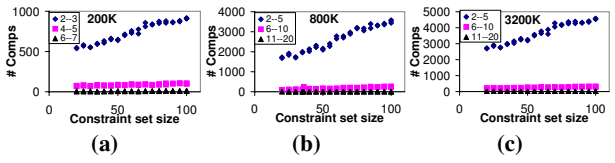


Figure 8: Component size distributions for datasets (a) 200K, (b) 800K (real Automotive), (c) 3200K

We draw the following two conclusions from these results. First, as constraint set size increases, the number of modestly sized components remains small. Only the number of extremely small components (with between 2 - 5 imprecise facts each) increases noticeably. Thus, the total number of configurations enumerated by the ProcessComponents step for all connected components in the hypergraph remain reasonable as constraint set size increases.

7.3 Regularity Experiments

One way the Marginalization algorithm achieves algorithmic efficiency is by exploiting constraint-space regularity, which the next set of experiments explores.

7.3.1 Constraint Partitioning

The next set of experiments examines how constraint-space regularity affects the distribution of connected component sizes. For the constraint language we propose in Section 4.1, the "head" of a constraint $A \implies B$ partitions the facts into non-overlapping sets such that each facts in each hyperedge are drawn from the same set.

For example, consider a constraint "two facts with the same location must have the same brand." We can think of this constraint as first partitioning all facts by their *Location* value. Then, each hyperedge introduced for this constraint contains only a set of facts in the same *Location* partition (e.g., facts which may have same *Location*, but potentially have different values for *Brand*). Since hyperedges only contain facts within the partition, the resulting component can be no larger than the partition regardless of the number of hyperedges present. *Essentially, the distribution of these partition sizes represents the worst-case distribution of component sizes.*

Our next set of experiments investigates the distribution of partition sizes as constraint set size increases, and is similar to the component size distribution experiments in Section 7.2. The same 35 constraint sets were used for these experiments and the results are given in Figures 9.a-c. Each curve in a graph gives the number of partitions with the indicated number of imprecise facts as the constraint set size varies. E.g., the curve labelled "2 – 5" indicates the number of partitions with between 2 and 5 imprecise facts. The partitions are created by grouping together facts which satisfy the head of the constraint, and the reported sizes are the number of imprecise facts in the partitions. Constraint set size k was defined in Section 7.1. We omit the curves for the range 0 to 1 imprecise facts for reasons similar to the ones given for the component size results (Figure 8).

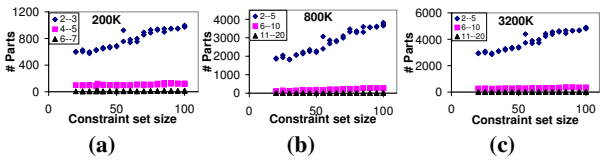


Figure 9: Partition size distributions for datasets (a) 200K, (b) 800K (real Automotive), (c) 3200K

By directly comparing between these figures and the corresponding results for component size distribution in Figures 8.a-c, we see the observed component size distributions are quite similar to the partition size distributions. Thus, we conclude that for the datasets

and constraint sets we consider, the component distribution is close to worst-case. Despite this worst-case behavior, the total number of configurations ProcessComponents enumerates remains reasonable.

7.3.2 Graph Connectivity

The purpose of this experiment is to directly measure the impact of the Non-Adjacent set optimization on running time. For this experiment, we used the Automotive dataset and generated a collection of related constraint sets so that we could control the connectivity of the constraint graph, which we define as the percentage of possible edges within a constraint hypergraph that are actually present. Intuitively, if the non-adjacent set optimization was not used, the number of configurations enumerated by ProcessComponent would be the same regardless of the hypergraph connectivity (i.e., all possible configuration for each component would have to be enumerated), thus increasing the running time of ProcessComponent. The results in Figure 10 indicate the Non-Adjacent set optimization significantly impacts running time if the hypergraph connectivity is low. For the real-world Automotive dataset, the constraint sets we consider resulted in constraint hypergraphs with very low connectivity (under 10% in many cases).

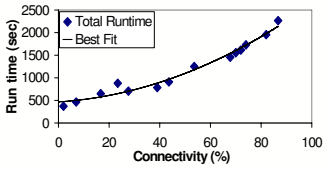


Figure 10: Regularity Experiment

7.4 Summary of Experimental Results

Together, the results in Sections 7.2 and 7.3 explain the scalability and efficiency of our approach, as observed in Section 7.3.1. The favorable distribution of connected component sizes we observed in Section 7.2 is a result of constraint-space regularity leading to many trivially small connected components. Second, the non-adjacent set optimization significantly reduces the number of configurations which ProcessComponents enumerates, and in practice significantly improves the performance of the Marginalization Algorithm. We note that further experimentation on additional datasets from other domains is required to determine the generality of these results.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have significantly extended the framework for OLAP over imprecise data presented in [8], to support domain constraints. This extension removes the strong independence assumptions required by [8], which are often violated in practice.

There are several interesting directions for further study. First, the constraint language we propose could be generalized to support more expressive types of constraints, similar in spirit to the ones proposed in [15]. For example, the language could be extended to support constraints over aggregation query results in each possible world (e.g., "The sum of all Sales in a world must be $1000"). A second related direction would be to develop a *less* expressive (but still useful) constraint language that would allow for development of more efficient marginalization algorithms, with the simpler language having additional regularity in the constraint-space that the algorithm can exploit. Finally, it would be fruitful to develop incremental maintenance algorithms for the MDB $D*$, similar in spirit to the EDB maintenance algorithms proposed in [7]. Incremental maintenance of the MDB is challenging since support is required for both updates to the underlying fact table and the set of constraints C.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Trans. Database Syst.*, 15(3):427–458, 1990.

[3] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. *In ICDE 2006*.

[4] L. Antova, C. Koch, and D. Olteanu. $10\hat{(}10\hat{6})$ worlds and beyond: Efficient representation and processing of incomplete information. *In ICDE 2007*.

[5] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theor. Comput. Sci.*, 296(3):405–434, 2003.

[6] L. Bertossi. Consistent query answering in databases. *SIGMOD Rec.*, 35(2):68–76, 2006.

[7] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Efficient allocation algorithms for OLAP over imprecise data. *In VLDB 2006*.

[8] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. *In VLDB 2005*.

[9] D. Burdick, A. Doan, R. Ramakrishnan, and S. Vaithyanathan. OLAP over imprecise data with domain constraints. UW-Madison CS Dept tech report cs-tr-1595. 2007.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[11] J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. *In CIKM 2004*.

[12] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *In VLDB 2004*.

[13] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. *In SIGMOD 1994*.

[14] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web community portals: A top-down, compositional, and incremental approach. *In VLDB 2007*.

[15] S. Flesca, F. Furfaro, and F. Parisi. Consistent query answers on numerical databases under aggregate constraints. *In DBPL 2005*.

[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *DMKD*, 1(1):29–53, 1997.

[17] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. *In VLDB 2006*.

[18] M. Halldorsson. Approximations of weighted independent set and hereditary subset problems. *Journal of Graph Algorithms and Applications*, 4(1):1–16, 2000.

[19] J. Y. Halpern. *Reasoning about Uncertainty*. MIT Press, Cambridge, MA, USA, 2003.

[20] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[21] C. A. Hurtado and A. O. Mendelzon. Olap dimension constraints. *In PODS 2002*.

[22] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[23] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1):40–48, 2006.

[24] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[25] J. Lechtenbörger, H. Shu, and G. Vossen. Aggregate Queries Over Conditional Tables. *JIIS*, 19(3):343–362, 2002.

[26] H.-J. Lenz and A. Shoshani. Summarizability in olap and statistical data bases. *In SSDBM 1997*.

[27] M. Rafanelli. *Multidimensional Databases: Problems and Solutions*. Idea Group Inc (IGI), 2003.

[28] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. In *Principles and Practice of Constraint Programming 1994*.

[29] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. UM Computer Science Dept Tech Report CS-TR-4820. 2006.

[30] M. Vardi. On the integrity of databases with incomplete information. *In PODS 1986*.

[31] J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.

[32] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. *In CIDR 2005*.

[33] C. Zaniolo. Database relations with null values. *J. Comput. Syst. Sci.*, 28(1):142–166, 1984.