Efficient Allocation Algorithms for OLAP over Imprecise Data

Doug Burdick[1*] Prasad M. Deshpande[3] T.S. Jayram[2]
Raghu Ramakrishnan[4] Shivakumar Vaithyanathan[2]

[1]University of Wisconsin, Madison [2]IBM Almaden Research Center
[3]IBM India Research Lab, SIRC [4]Yahoo! Research

ABSTRACT

Recent work proposed extending the OLAP data model to support data ambiguity, specifically imprecision and uncertainty. A process called allocation was proposed to transform a given imprecise fact table into a form, called the Extended Database, that can be readily used to answer OLAP aggregation queries.

In this work, we present scalable, efficient algorithms for creating the Extended Database (i.e., performing allocation) for a given imprecise fact table. Many allocation policies require multiple iterations over the imprecise fact table, and the straightforward evaluation approaches introduced earlier can be highly inefficient. Optimizing iterative allocation policies for large datasets presents novel challenges, and has not been considered previously to the best of our knowledge. In addition to developing scalable allocation algorithms, we present a performance evaluation that demonstrates their efficiency and compares their performance with respect to straightfoward approaches.

1. INTRODUCTION

OLAP is based on the multidimensional model of data, in which attributes of facts are of two types, *dimensions* and *measures*, and facts can be seen as points in a corresponding multidimensional space. If we relax the assumption that all facts are points, and allow some facts to be regions (consistent with the domain hierarchies associated with dimension attributes), we must deal with the resulting imprecision when answering queries. For example, we can denote that a particular repair took place in the state Wisconsin, without specifying a city. Dealing with such uncertain information is widely recognized to be an important problem and has received increasing attention recently.

In [5], we proposed a possible-worlds interpretation of imprecision that leads to a novel allocation-based approach to defining semantics for aggregation queries. Operationally, *allocation* is performed by replacing each imprecise "region" fact r in database D with a set of precise "point" facts representing the possible completions of r. Each possible completion is assigned an *allocation weight*, and any procedure for assigning these weights is referred to as an *allocation policy*. The result of applying an allocation policy to an imprecise database D is referred to as an *extended database*.

Allocation is motivated and justified in [5] as a mathematically principled method for handling imprecision, with a general framework for characterizing the space of allocation policies detailed in [6]. However, neither work explored scalability and performance of allocation algorithms. Designing scalable allocation algorithms is a challenge because of the complex relationships between the precise and imprecise facts that need to be considered while performing allocation. Additionally, several of the allocation policies are iterative in nature, making the costs of allocation prohibitive unless specialized algorithms are developed.

In this paper, we consider the computational aspects of allocation policies and present scalable allocation algorithms. Our contributions can be summarized as follows:

1. Abstraction of various allocation policies into a policy template. Scalable algorithms developed for the template are thus applicable to the entire set of allocation policies.

2. Capturing the relationships between the precise and imprecise facts in terms of an allocation graph formalism, which provides a basis for developing allocation algorithms.

3. Present a set of scalable algorithms that optimize the I/Os and sorts required to perform allocation, including an algorithm that exploits the connected components of the allocation graph to optimize across iterations. For iterative allocation policies, this turns out to be very important.

4. We present an efficient algorithm for maintaining the Extended Database to reflect updates to the given fact table.

5. An experimental evaluation of the performance and scalability of the proposed algorithms.

The rest of this paper is organized as follows. In Section 2, we review some of the definitions and notations used in [5, 6]. In Section 3, we present a framework that abstracts the common elements of various allocation policies. In Sections 4 - 6, we consider locality issues and present the Independent and Block algorithms. In Sections 7 and 8, we consider issues related to iteration and present the Transitive algorithm. In Section 9, we present an efficient algorithm for maintaining the Extended Database when the imprecise fact table is updated. We discuss some related work in Section 10 and present our experimental evaluation in Section 11.

[*]Work performed while author visiting IBM Almaden Research Center

VLDB '06, September 12-15, 2006, Seoul, Korea.

2. NOTATION AND BACKGROUND

In this section, our notation is introduced and the problem is motivated using a simple example.

2.1 Data Representation

Attributes in the standard OLAP model are of two kinds— *dimensions* and *measures*. Each dimension in OLAP has an associated hierarchy, e.g., the location dimension may be represented using City and State, with State denoting the generalization of City. In [5], the OLAP model was extended to support imprecision in dimension values that can be defined in terms of these hierarchies. This was formalized as follows.

Definition 1 (Hierarchical Domains)**.** A *hierarchical* domain H over base domain B is a power set of B such that (1) $\emptyset \notin H$, (2) H contains every singleton set (i.e., corresponds to some element of B), and (3) for any pair of elements $h_1, h_2 \in H$, $h_1 \supseteq h_2$ or $h_1 \cap h_2 = \emptyset$. Non-singleton elements of H are called *imprecise* values. For simplicity, we assume there is a special imprecise value ALL such that $h \subseteq$ ALL for all $h \in H$.

Each element $h \in H$ has a *level*, denoted by $\text{LEVEL}(h)$, given by the number of elements of H (including h) on the longest chain (w.r.t. $\subseteq$) from h to a singleton set. □

Intuitively, an imprecise value is a non-empty set of possible values. Hierarchical domains impose a natural restriction on specifying this imprecision. For example, we can use the imprecise value Wisconsin for the location attribute in a data record if we know that the sale occurred in the state of Wisconsin but are unsure about the city. Each singleton set in a hierarchical domain is a leaf node in the domain hierarchy and each non-singleton set is a non-leaf node. For example, Madison and Milwaukee are leaf nodes whose parent Wisconsin is a non-leaf node. The nodes of H can be partitioned into level sets based on their level values, e.g. Madison belongs to the 1st level whereas Wisconsin belongs to the 2nd level. The nodes in level 1 correspond to the leaf nodes, and the element ALL is the unique element in the highest level.

Definition 2 (Fact Table Schemas and Instances)**.** A *fact table schema* is $\langle A_1, A_2, \ldots, A_k; L_1, L_2, \ldots, L_k; M_1, M_2, \ldots, M_n \rangle$ such that (i) each dimension attribute A_i, $i \in 1 \ldots k$, has an associated hierarchical domain, denoted by $\text{dom}(A_i)$, (ii) each level attribute L_i, $i \in 1 \ldots k$ is associated with the level values of $\text{dom}(A_i)$, and (iii) each measure attribute $M_j, j \in 1 \ldots n$, has an associated domain $\text{dom}(M_j)$ that is either *numeric* or *uncertain*.

A *database instance* of this fact table schema is a collection of *facts* of the form $\langle a_1, a_2, \ldots, a_k; \ell_1, \ell_2, \ldots \ell_k; m_1, m_2, \ldots, m_n \rangle$ where $a_i \in \text{dom}(A_i)$ and $\text{LEVEL}(a_i) = \ell_i$, for $i \in 1 \ldots k$, and $m_j \in \text{dom}(M_j)$, $j \in 1 \ldots n$. □

Definition 3 (Cells and Regions)**.** Consider a fact table schema with dimension attributes $A_1, \ldots, A_k$. A vector $\langle c_1, c_2, \ldots, c_k \rangle$ is called a *cell* if every c_i is an element of the base domain of A_i, $i \in 1 \ldots k$. The *region* of a dimension vector $\langle a_1, a_2, \ldots, a_k \rangle$, where $a_i \in \text{dom}(A_i)$, is defined to be the set of cells $\{\langle c_1, c_2, \ldots, c_k \rangle \mid c_i \in a_i, i \in 1 \ldots k\}$. Let $\text{reg}(r)$ denote the mapping of a fact r to its associated region. □

Since every dimension attribute has a hierarchical domain, we thus have an intuitive interpretation of each fact in the database being mapped to a region in a k-dimensional space. If all a_i are leaf nodes, the fact is *precise*, and describes a region consisting of a single cell. Abusing notation slightly, we say that the precise fact is mapped to a cell. If one or more A_i are assigned non-leaf nodes, the fact is *imprecise* and describes a larger k-dimensional region.

FactID	*Loc*	*Auto*	LocL	AutoL	*Sales*
p1	MA	Civic	1	1	100
p2	MA	Sierra	1	1	150
p3	NY	F150	1	1	100
p4	CA	Civic	1	1	175
p5	CA	Sierra	1	1	50
p6	MA	Sedan	1	2	100
p7	MA	Truck	1	2	120
p8	CA	ALL	1	3	160
p9	East	Truck	2	2	190
p10	West	Sedan	2	2	200
p11	ALL	Civic	3	1	80
p12	ALL	F150	3	1	120
p13	West	Civic	2	1	70
p14	West	Sierra	2	1	90

Table 1: Sample data

Each cell inside this region represents a possible completion of an imprecise fact, formed by replacing non-leaf node a_i with a leaf node from the subtree rooted at a_i.

Example 1. Consider the fact table shown in Table 1. The first two columns are dimension attributes *Location* (*Loc*) and *Automobile* (*Auto*), and take values from their associated hierarchical domains. The structure of these domains and the regions of the facts are shown in Figure 1. The sets State and Region denote the nodes at levels 1 and 2, respectively, for *Location*; similarly, Model and Category denote the level sets for *Automobile*. The next two columns contain the level-value attributes *Location-Level* (*LocL*) and *Automobile-Level* (*AutoL*), corresponding to *Location* and *Automobile* respectively. For example, consider fact p6 for which *Location* is assigned MA, which is in the 1st level, and *Automobile* is assigned Sedan, which is in the 2nd level. These level values are the assignments to *Location-Level* and *Automobile-Level*, respectively.

Precise facts, p1–p5 in Table 1, have leaf nodes assigned to both dimension attributes and are mapped to the appropriate cells in Figure 1. Facts p6–p14, on the other hand, are imprecise and are mapped to the appropriate multidimensional region. For example, fact p6 is imprecise because the *Automobile* dimension is assigned to the non-leaf node Sedan and its region contains the cells (MA,Camry) and (MA,Civic), which represent possible completions of $p6$. □

3. FRAMEWORK FOR ALLOCATION

In this section, we quickly review the basic framework for allocation policies. First, we restate the general template for allocation policies presented previously in [6]. Then, we present a graph-based framework to conceptualize the flow of data required to perform allocation.

3.1 Allocation Policies

For completeness we restate the following definition from [5, 6].

Definition 4 (Allocation Policy and Extended Data Model)**.** Let r be a fact in the fact table, and $reg(r)$ the region of r. For each cell $c \in \text{reg}(r)$, the *allocation* of fact r to cell c, denoted by $p_{c,r}$, is a non-negative quantity denoting the weight of completing r to cell c. We require that $\sum_{c \in reg(r)} p_{c,r} = 1$. An *allocation policy* A is a procedure that takes as its input a fact table consisting of imprecise

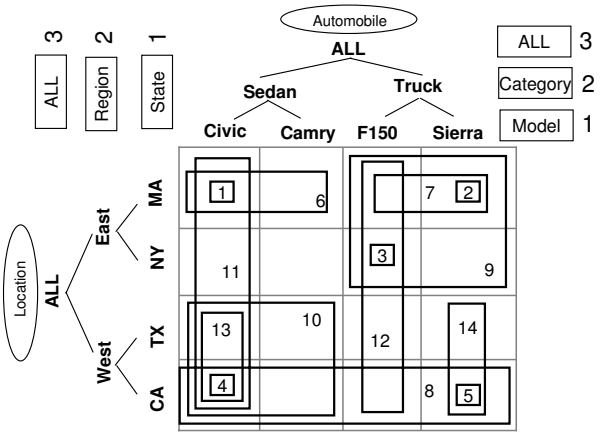

Figure 1: Multidimensional View of the Data

facts and produces as output the allocations of all the imprecise facts in the table. The result of applying such a policy to a database D is an *extended database* D^*. The schema of D^*, referred to as the *Extended Data Model(EDM)*, contains all the columns of D plus additional columns to keep track of the cells that have strictly positive allocations. Suppose that fact $r \in D$ has a unique identifier denoted by $\mathrm{ID}(r)$. Corresponding to each fact $r \in D$, we create a set of fact(s) $\langle \mathrm{ID}(r), r, c, p_{c,r} \rangle$ in D^* for every $c \in reg(r)$ such that $p_{c,r} > 0$ and $\sum p_{c,r} = 1$. Observe, each precise fact has a single allocation of 1 for the cell to which it maps. □

3.2 Allocation Policy Template

In [6], we demonstrated how the space of allocation policies considered in [5] can be mapped to the following *allocation policy template*, which is presented below. Each allocation policy instantiates this template by selecting a particular *allocation quantity* that will be used to assign the allocation weights. For example, EM-Count allocation (from [6]) uses fact count as the allocation quantity. The template is instantiated with $\Delta^t(c)$ equal to the "count" of facts which map to cell c (i.e., the sum of $p_{c,r}$ values for all facts r with non-zero allocation to cell c).

The selection of an allocation quantity corresponds to making an assumption about the correlation structure present in the data that should be reflected in the assignment of allocations, and details are provided in [6].

Definition 5 (Allocation Policy Template)**.** Assume allocation policy A has been selected, which determines the associated allocation quantity. For each cell c, let $\delta(c)$ be the value of the allocation quantity assigned to c. Let $\Delta^t(c)$ be the updated quantity assigned to c during iteration t to account for all imprecise facts r overlapping c. Let $\Gamma^{(t)}(r)$ denote the quantity associated with fact r. Then, for an imprecise fact table D, the set of *update equations* are generated from the following template:

$$\Gamma^{(t)}(r) \quad = \quad \sum_{c' :\ c' \in \mathrm{reg}(r)} \Delta^{(t-1)}(c') \qquad (1)$$

$$\Delta^{(t)}(c) \quad = \quad \delta(c) + \sum_{r :\ c \in \mathrm{reg}(r)} \frac{\Delta^{(t-1)}(c)}{\Gamma^{(t)}(r)} \qquad (2)$$

For each cell c that is a possible completion of fact r, the allocation of r to c is given by $p_{c,r} = \Delta^{(t)}(c)/\Gamma^{(t)}(r)$ □

For a given imprecise fact table D, the collection of update equations is specified by instantiating this template with the appropriate quantities for each fact r and cell c. Every imprecise fact $r \in D$ has an equation for $\Gamma^{(t)}(r)$, and likewise every cell $c \in C$ has an equation $\Delta^{(t)}(c)$.

Observe the equations generated by this framework are iterative, as denoted by the superscripts. The equations in the above template can be viewed as defining an *Expectation Maximization (EM)* framework (see [5, 6] for the details). Expression 1 of the template encodes the E-step (Expectation) and Expression 2 is the M-step (Maximization). In numerical EM, each $\Delta(c)$ is evaluated iteratively until the values between successive iterations stop changing (i.e., the value converges). Formally, let $\epsilon = \frac{|\Delta^{(t)}(c) - \Delta^{(t+1)}(c)|}{\Delta^{(t)}(c)}$. If $\epsilon < k$, where k is a pre-determined constant, then we say the value for $\Delta(c)$ has *converged*. When $\Delta(c)$ for all cells c have converged, the iteration stops. At this point, the final allocation weights $p_{c,r}$ are available.

Further details regarding the mathematical justification for this space of iterative allocation policies is covered in [5, 6], and will not be revisited in this work. However, we will describe the intuition behind such iterative allocation policies. Allocation policies should take into account interactions between overlapping imprecise facts. Consider imprecise facts $p11$ and $p6$ from the example in Figure 1. Intuitively, the allocation of $p11$ should affect the allocation of $p6$, and symmetrically, the allocation of $p6$ should affect the allocation of $p11$ since these facts overlap. However, it should be clear that different allocation weights are obtained for the completions of facts $p6$ and $p11$ in the EDB depending on the relative order in which the facts are allocated. Iterative allocation policies avoid this issue because they will converge to the same allocation weights regardless of the order in which the facts are allocated. Thus, allocation can be considered a set-based operation if iterative allocation policies are used.

3.3 Allocation Graph and Basic Algorithm

The template given above only enumerates the set of allocation equations, and provides no insight into the operational aspects regarding their evaluation. For example, the required access patterns of cell data C and imprecise facts I in D are not clear, and such information is necessary for designing efficient, scalable algorithms. To address this, we present an operational framework using a bipartite graph-based formalism, called *allocation graph*.

Definition 6 (Allocation Graph)**.** Assume allocation policy A has been selected to handle imprecision in fact table D. Let I denote the set of imprecise facts in D and C denote the set of cells representing possible completions of facts in I, as determined by A.

The *allocation graph* of D (w.r.t. A) is defined as follows: Each cell $c \in C$ corresponds to a node shown on the left side of Figure 2, while each imprecise fact in $r \in I$ corresponds to a node shown on the right side. There is an edge (c, r) in G if and only if c is a possible completion of r. (i.e., $c \in \mathrm{reg}(r)$). □

In the above definition, the set of cells C depends on the selected allocation policy A, and is *not equivalent* to the set of precise facts in D. The values of $\delta(c)$ for each entry c may be determined from the precise facts, but this is not required. For example, each allocation policy in [5, 6] used one of the following choices: the set of cells mapped to by at least one precise fact from D, the union of the regions of the imprecise facts, or the cross product of base domains for all dimensions (i.e., every possible cell). Regardless of the choice for C made by A, the allocation graph formalism can still be used. The allocation graph for the sample data in Table 1 (w.r.t. EM-Count allocation policy) is given in Figure 2.

Notice that the allocation graph is bipartite. We now present an *allocation algorithm template* called the Basic Algorithm that de-

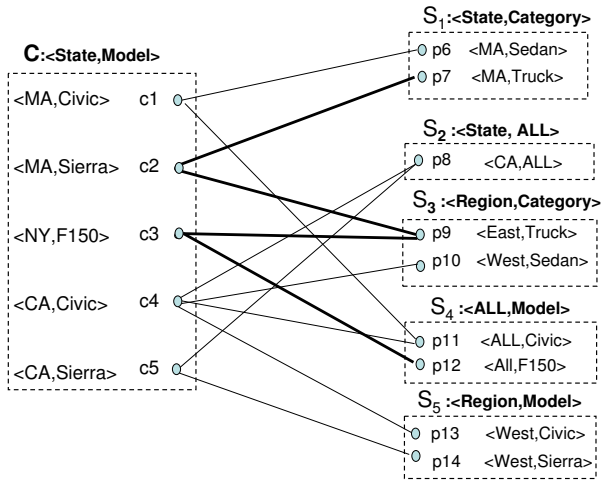

Figure 2: Allocation graph for data in Table 1

scribes how to evaluate the collection of allocation equations generated by A in terms of processing these edges in G (i.e., processing terms in the allocation equations). The pseudocode is listed in Algorithm 1.

Theorem 1. *For a given imprecise fact table D and selected allocation policy A, let G be the resulting allocation graph for D (w.r.t. A). The processing of edges in G performed by the Basic Algorithm is equivalent to evaluating the collection of allocation equations generated by A.* □

Proof. By construction, G contains an edge (c, r) between cell c and imprecise fact r if and only if c is a possible completion of r. In terms of the set of allocation equations, each edge $(c, r) \in G$ corresponds to exactly one term in both the $\Gamma^{(t)}(r)$ equation for fact r and the $\Delta^{(t)}(c)$ equation for cell c. Consider lines 6 – 9 of the Basic Algorithm. It should be clear these loops visit each edge in G exactly once. This is equivalent to evaluating the corresponding term in a $\Gamma^{(t)}(r)$ equation for each imprecise fact r exactly once, which is correct.

Similarly, lines 11 – 14 correspond to evaluating $\Delta^{(t)}(c)$ equations for all cells c, with the processing of edges and update in lines 13 – 14 corresponding to evaluating the equation for cell c (generated from allocation template Equation 2). □

Thus, processing edges in G is equivalent to evaluating these equations. Notice each iteration t requires two passes over the edges of G, and during each pass, each edge of G is processed exactly once. Moreover, these passes cannot be replaced by a single pass because the second pass uses values computed for $\Gamma^{(t)}$ in the first pass of the *current* iteration t to update the values for $\Delta^{(t)}$ in the second pass.

3.4 Scalability of The Basic Algorithm

As presented, the Basic Algorithm has several issues scaling to large fact tables (i.e., fact tables such that C and I are larger than main memory). From the pseudocode in Algorithm 1, the nested loops in lines 6 – 9 require for each imprecise fact r access to all cells r overlaps. Similarly, the nested loops in lines 11 – 14 require access to all imprecise facts $r \in I$ overlapping cell c for each cell c. In general, there exists no ordering of either C or I providing the necessary locality Basic requires for either set. We refer to this problem as the *locality issue*.

Algorithm 1 Basic Algorithm

```
1:  Input: Allocation graph G with cells C and imprecise facts I
2:  for (each cell c) do
3:      Δ^(0)(c) ← δ(c)
4:  for (each iteration t until all Δ^(t)(c) converge) do
5:      // Compute t-th step estimate for Γ's
6:      for (each imprecise fact r) do
7:          Γ^(t)(r) ← 0
8:          for (each cell c s.t. edge (c, r) ∈ G) do
9:              Γ^(t)(r) ← Γ^(t)(r) + Δ^(t−1)(c)
10:     // Compute t-th step estimate for Δ's
11:     for (each cell c) do
12:         Δ^(t)(c) ← δ(c)
13:         for (each imprecise fact r s.t. edge (c, r) ∈ G) do
14:             Δ^(t)(c) ← Δ^(t)(c) + Δ^(t−1)(c)/Γ^(t)(r)
```

A second orthogonal issue arises from the iterative nature of the allocation algorithm. Assume a "good" ordering of the cell data C and imprecise facts I addressing the locality issue were available. Even then, both C and I need to be scanned completely *for each iteration* to execute the Basic Algorithm. This issue, which we refer to as the *iterative issue*, is significant in practice, since a non-trivial number of iterations are required before the allocation algorithm completes (i.e., the allocation weights converge)

The approaches presented to address the locality issue in Section 4 are incorporated into the Independent (Section 5) and Block algorithms (Section 6). Section 7 details our solution to the iterative issue, which serves as the basis for the Transitive Algorithm presented in Section 8.

4. ADDRESSING THE LOCALITY ISSUE

In this section, we present strategies addressing the locality issue which serve as the basis for creating I/O aware variants of the Basic Algorithm. In the pseudocode, listed in Algorithm 1, notice each iteration involves two passes over all edges in allocation graph G (i.e., one pass for the nested loops in lines 6 – 9 and a second for the nested loops in lines 11 – 14.) Addressing the locality issue involves carefully ordering the computations for each pass. In terms of G, this could be considered determining the best order for processing edges in G. We first consider whether we can partition the imprecise facts in some clever manner so that each group of imprecise facts can be processed separately within each pass. Before we study what partitions lead to efficient I/O computations, we first address the correctness of the proposed approach.

Theorem 2 (Ordering Of Edges)**.** *Suppose the update equation for $\Delta^{(t)}(c)$ is computed using a operator that is commutative and associative (e.g., sum). Let P be a partitioning of the edges of G into s subgraphs $G_1, G_2, \ldots, G_s$.*

Then, the final values for $\Delta^{(t)}(c)$ and $\Gamma^{(t)}(r)$ are unaffected by: 1) the choice of partitioning P, 2) the order in which subgraphs are processed or 3) the order in which edges within a subgraph are processed. □

The above theorem shows that we are free to choose any partitioning of the imprecise facts into groups, and can arrive at the same result. Pseudocode for a variant of the Basic Algorithm utilizing this partitioning concept, called *Partitioned Basic*, is given in Algorithm 2. Observe the nested loops in lines 11 – 12 iterate over cells then records. From the result in Theorem 2, this ordering is permissable as long as each edge in G is visited exactly once. For ease of presentation, details regarding initialization and the update equation have been omitted.

Corollary 1. *From Theorem 1 and Theorem 2, the Partitioned Basic Algorithm computes the same results as the Basic Algorithm.* □

Algorithm 2 Partitioned Basic Algorithm

```
1: Input: Allocation graph G with cells C and imprecise facts I
2: Input: Partitioning P_1, P_2, ... P_s of the imprecise facts I
3: for (each iteration t until all Δ^(t)(c) converge) do
4:    // Compute t-th step estimate for Γ
5:    for (each partition P_i) do
6:       for (each cell c) do
7:          for (each record r in P_i s.t. (c, r) ∈ G_i) do
8:             // Update Γ^(t)(r)
9:    // Compute t-th step estimate for Δ
10:   for (each partition P_i) do
11:      for (each cell c) do
12:         for (each record r in P_i s.t. (c, r) ∈ G_i) do
13:            // Update Δ^(t)(c)
```

4.1 Summary Tables

In order to study appropriate partitions of the imprecise facts for the Partitioned Basic Algorithm, it will be helpful to group together imprecise facts according to the levels at which the imprecision occurs. We formalize this notion below.

Definition 7 (Summary Tables)**.** Fix an allocation graph G, and let I be the set of imprecise facts and C be the set of cells. Partition the facts in I by grouping together facts in I that have an identical assignment for the vector of level attributes. We refer to each such grouping of the imprecise facts as a *summary table*. Note that each summary table is associated with a distinct assignment to the level attributes. Since all cells in C correspond to the lowest level of the dimensional hierarchies, for convenience we refer to C as the *cell summary table*. □

Intuitively, the summary tables are "logical" groupings which are similar to the result of performing a Group-By query on the level attributes. The main difference is that summary tables only contain entries corresponding to either imprecise facts in D or cells in C. As a consequence, there is a partial ordering between summary tables similar to the one between Group-By views, described in [14].

Definition 8 (Partial Ordering of Summary Tables ($\preceq$))**.** Let $\mathcal{S}$ be the collection of summary tables for D. The vector of level values for summary table S_i, referred to as *level-vector*, is denoted as $level(S_i)$ (i.e., all facts in S_i have level-vector $level(S_i)$). Then, for each $S_i, S_j \in \mathcal{S}$, $S_i \preceq S_j$ iff for each position p in $level(S_i)$, $level(S_i)_p < level(S_j)_p$ and there does not exist any $S_k \in \mathcal{S}$ such that $S_i \preceq S_k \preceq S_j$. □

We note that that $\preceq$ is transitive, but not closed since $\mathcal{S}$ does not include every possible summary table.

Since each summary table is associated with a unique level vector, it is possible to materialize the separate summary tables using a single sort. The sorting key is formed by concatenating the level and dimension attributes. This "special sort", which we refer to as *sorting D into summary table order*, can be thought of as simultaneously accomplishing the following: 1) partition the precise and imprecise facts, 2) process the precise facts to materialize C (i.e., determine $\delta(c)$ for each $c \in C$, and 3) further partition the imprecise facts into the separate summary tables. In the descriptions of the algorithms that follow, we assume this pre-processing step has been performed. In terms of I/O operations, it is equivalent to sorting D.

Example 2. Consider the sample data in Table 1, with the EM-Count allocation policy. For this data set, there are 6 summary tables—the cell summary table C and 5 imprecise ones $S_1, \ldots, S_5$—as indicated by labels for each of the tables in Figure 3. The multi-dimensional representation for each summary table is shown. Each summary table is labeled by the level-vector associated with that table. For example, the summary table (State,Category) consists of all facts whose level-vector equals $< 1, 2 >$. Notice the entries in C are *not* precise facts, but correspond to *cells*. □

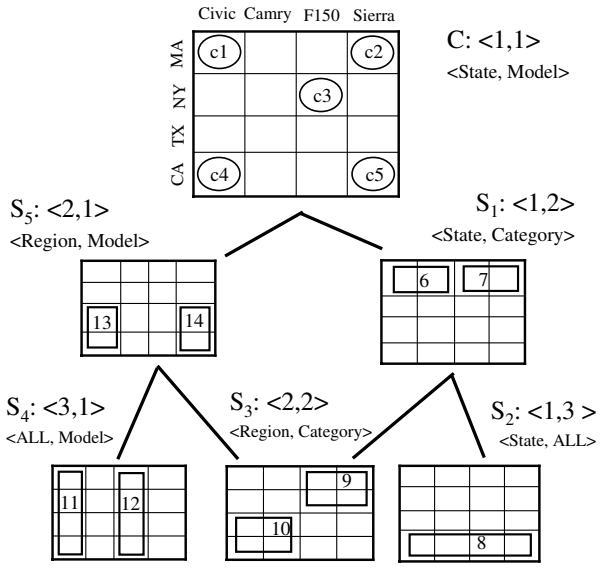

Figure 3: Summary Tables for Example Data (with partial order indicated)

Why are summary tables important in the context of the Partitioned Basic Algorithm? The answer is that computing a single pass for each summary table S_i can be achieved using *one scan* of S_i and C, as shown below.

Theorem 3. *For every imprecise summary table S_i, there exists a sort of S_i and the cell summary table C such that a single pass through the edges of the subgraph between C and S_i can be executed using a single scan of C and S_i.* □

The proof of the above theorem relies on the fact that the above subgraph has a simple structure: every cell c is overlapped by at most one imprecise fact in S_i. Since the degree of each cell c is at most 1 in the subgraph between C and S_i, it is possible to order corresponding entries in C and S_i so that for every imprecise fact $r \in I$, cells overlapped by r (i.e., nodes adjacent to r in G) form contiguous blocks, and these blocks are pairwise disjoint across the imprecise facts. The sort order can be achieved by sorting on a key formed by concatenating together the level and dimension attribute vectors. The Independent algorithm described in the next section builds on this idea by considering sort orders that are consistent with multiple summary tables so that bigger groupings of imprecise facts are possible.

4.2 Partitions

What happens when the sort order of C is not consistent with the imprecise summary table? In this case, we no longer have pairwise disjoint contiguous blocks. This is easily seen in the allocation graph in Figure 2. Let the order on the cells be from top to bottom as shown in the figure, and consider summary table S_4 with

imprecise facts $p11$ and $p12$. The cells adjacent to $p11$ are $c1$ and $c4$. However, any contiguous block including these two cells also contains cell $c3$, which is adjacent to $p12$, and Thus, it appears we have to re-sort C to process S_4. However, if enough space in memory were available to simultaneously hold all imprecise facts in S_4 whose processing has not been finished, it is still possible to use the current sort order. We now formalize this intuition.

Definition 9 (Partition Size)**.** Let C be a cell summary table sorted with respect to some sort order L and let S_i be a summary table. We say that the division of cells in C into contiguous blocks (i.e., respecting the sort order) is *legal* if for every imprecise fact, all of its neighbors in G are within exactly one of the contiguous blocks. The *partition size* of S_i with respect to the sort order L on C is the largest number of facts that map to the same contiguous block of cells given the best legal division of cells into contiguous blocks, i.e., this number must be as small as possible. □

Theorem 4. *Let C be a cell summary table sorted with respect some sort order L (i.e., ordering of the values in the level and dimension attribute vectors) and let S_i be a summary table. Then a single pass on the subgraph between C and S_i can be executed using a single scan of C and S_i provided that the memory available is as large as the partition size of S_i with respect to sort order L on C.* □

Thus, the partition size of summary table S_i is the largest amount of memory that needs to be reserved for processing S_i in a single pass, and depends on the chosen sort order of the dimensions L. We make the observation that the partition size for each S_i can be computed during the step where D is sorted into summary table order as follows.

Consider summary table S_i. During the final "merging step" of the sort into summary table order, each consecutive pair of entries r_1, r_2 in the final sorted S_i are compared to determine their ordering in S_i. Before this comparison takes place, for each $r \in S_i$, we determine the smallest and largest indexes of entries in C such that edge $(c, r) \in G$. These are denoted $r.first$ and $r.last$ respectively. Observe that a partition boundary for S_i can only occur between consecutive entries r_1, r_2 in the final sorted order of S_i if $r_2.first > r_1.last$.

The inequality condition holding signifies that all edges have been visited for r_1 before the first edge of r_2 will be visited, and corresponds to the equation $\Gamma^{(t)}(r_1)$ being completely evaluated (i.e., all terms in the equations seen) before evaluation of $\Gamma^{(t)}(r_2)$ starts (i.e., first term in the equation is seen).

Example 3. Consider a "pathological" fact table similar to the running example, but which has every possible fact in each imprecise summary table and generates cell summary table C containing a $\delta(c)$ entry for all possible cells. Figure 4 shows the multidimensional representation of this new example fact table after it has been sorted into summary table order. Assume the sort order L is {Location, Automobile}, and that summary table entries are sorted in the order indicated by the labels on each entry. The sort order of the cells is $c1, \ldots, c16$.

From Theorem 4, *any* of the S_i can be processed in a single scan of both S_i and C if enough memory is available to hold the block of entries with the "thick" edges for each S_i. For example, S_1 and S_2 require 1 entry, S_2 and S_4 require 4 entries, and S_3 requires 2 entries. This number of required entries is the corresponding partition size for each S_i respectively. □

The Block algorithm, described in Section 6, exploits this idea by finding a single sort that can be used to process all summary tables

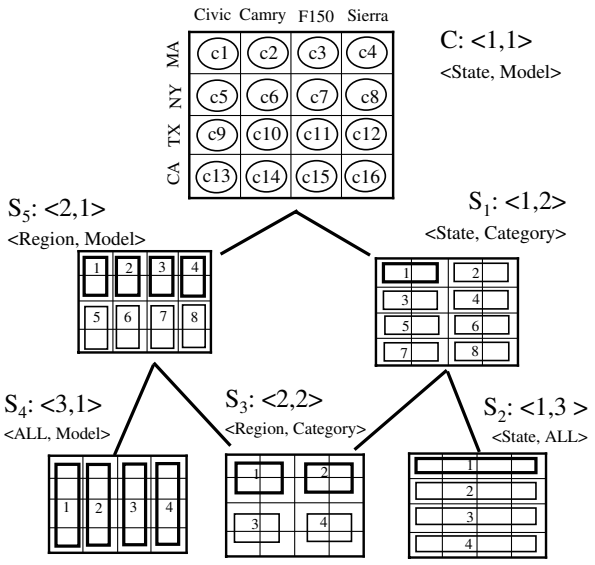

Figure 4: Illustrative Example of Determining Partition Sizes

in multiple scans, where each scan involves processing as many summary tables as possible whose total partition size fits within available memory.

5. INDEPENDENT ALGORITHM

In this section we introduce the *Independent* algorithm which improves upon the Partitioned Basic Algorithm by exploiting structure of the summary table partial order.

5.1 Summary Table Structure

We now re-consider the partial order between summary tables noted in Section 4.1. First, we generalize Theorem 3 to groups of summary tables.

Theorem 5. *Consider a path through the summary table partial order, containing in order summary tables $C \preceq S_1 \preceq S_2 \preceq \cdots \preceq S_k$. There exists a sort order L over all S_i in the path and the cell summary table C such that all edges in the subgraph of G between the S_i and C can be processed by executing a single simultaneous scan of the S_i and C.* □

After performing the step where D is sorted into summary table order, we have information about which imprecise summary tables have entries corresponding to facts from D, and can construct the summary table partial order. For a given summary table partial order, the result from [15] can be trivially adapted to provide a lower bound on the number of chains in the partial order, and to identify the tables in each chain as well. The lower bound is the length of the longest anti-chain in the summary table partial order (i.e., the "width" W), which is is the minimum number of sorts required of C. Given the summary tables in a chain, the results from [15] can be used to obtain the required sort order to process the chain.

5.2 Independent Details

The pseudocode for the Independent algorithm is given in Algorithm 3. For ease of presentation, the initialization steps are omitted, since they are identical to those described in the Basic Algorithm. We assume that D has been sorted into summary table order and summary table partial order information is available.

For each summary table in the chain (including the precise summary table C) we only need enough memory to hold a single record. Since we consider records in page-sized blocks, we actually perform I/Os for an entire page of records. However, we refer to the single current record for each summary table as the *summary table cursor for* S_i, which can be thought of as a pointer to a specific entry in the buffer for S_i. The pseudocode contains the step "update cursor on S_i to record r that could cover c." Details are implementation specific and involve examining the dimension attribute values of c and r to find single fact in S_i that covers c.

Corollary 2 (Correctness of Independent Algorithm)**.** *From Theorems 2 and 4, the Independent Algorithm computes the same results as the Partitioned Basic Algorithm.* □

Algorithm 3 Independent Algorithm

```
1:  Input: Cell-level summary table C, Imprecise Summary Table Groupings S, Sort-Order Listings L
2:  for (each iteration t until all Δ^(t)(c) converge) do
3:    for (each summary-table group S_g ∈ S) do
4:      Sort C and summary-tables in S_g into sort-order L_g
5:      // Compute t-th step estimate for Γ
6:      for (each cell c) do
7:        for (each summary table S_i ∈ S_g) do
8:          Update cursor on S_i to record r that could cover c
9:          if (r ≠ NULL) then
10:           Γ^(t)(r) ← Γ^(t)(r) + Δ^(t−1)(c)
11:   // Compute t-th step estimate for Δ
12:   for (each summary table group S_g ∈ S) do
13:     for (each cell c) do
14:       for (each summary table S_i ∈ S_g) do
15:         Update cursor on S_i to record r that could cover c
16:         if (r ≠ NULL) then
17:           Δ^(t)(c) ← Δ^(t)(c) + Δ^(t−1)(c)/Γ^(t)(r))
```

Following our convention, we omit the costs of sorting D into summary table order and the final cost of writing out the Extended Database D^*, since these are common to all algorithms.

Theorem 6. *Let $|C|$ be the size of C in pages and I the combined total size in pages of all imprecise summary tables. Let W be the length of the longest anti-chain in the summary table partial order, and T the number of iterations. The Independent Algorithm in the worst case requires $7T(W|C| + |I|)$ I/Os.* □

Proof. We make the standard assumption that external sort requires two passes over a relation, with each page being read and written during a pass. Each summary table in S_g and C are sorted into the corresponding sort-order of L_g. Then, two passes are required over each summary table in S_g and C. During the first pass, each page of C is read only, and during the second pass, each page of C is read and written. Thus, the two allocation passes require 3 I/Os per page in C. Similarly, each page in an imprecise summary table requires 3 I/Os: a read and write for the first pass, and only a read for the second pass.

The total number of required I/Os per iteration is given by the following expression. $\sum_{i=1}^{W}$[sort C + sort of each imprecise summary table in summary table group i + 2 scans of C] + [2 scans of each summary table in group i]

= $4W|C|$ I/Os + $4|I|$ I/Os + $3W|C|$ I/Os + $3|I|$ I/Os. It is a straightforward exercise to simplify this expression to the one given in the theorem. □

6. BLOCK ALGORITHM

In practice, the cost of repeatedly sorting the cell summary table C is likely to be prohibitive. In general, the number of cells in C will be much larger than the number of records in the imprecise summary tables combined. For the common case where C does not fit into memory, each sort of C is equivalent to reading and writing every page of C twice, or $4|C|$ I/Os.

What was the motivation for the repeated sorts used in Independent? During any given point of execution, we only need to keep in memory records of S_i for which we have seen at least one cell in C and may see at least one more. Re-sorting C for each summary table group (i.e., the set of summary tables on a path through the summary table partial order) reduced this to 1 record for each S_i and C.

Building on the intuition presented in Section 4.2, we observe that *any* summary table can be processed using the same sort order if we can hold partition size of S_i records in memory for each S_i. Conceptually, this is equivalent to increasing the size of the summary table cursor from a single record to a contiguous block of records, which we called the *partition of* S_i. Only a single partition of summary table S_i needs to be held in memory as we scan C. We note the partition size for C is always 1.

Algorithm 4 Block Algorithm

```
1:  Method: Block Algorithm
2:  Input:Cell-level summary table C, Imprecise Summary Table Groupings S, Allocation Policy A
3:  for (each iteration t until all Δ^(t)(c) converge) do
4:    for (each summary-table group S_g ∈ S) do
5:      for (each cell c in C) do
6:        for (each summary table S_i ∈ S_g) do
7:          Update cursor on S_i to partition p that could cover c
8:          Find record r in p that could cover c
9:          //If p contains such an r, perform allocation
10:         if (r ≠ NULL) then
11:           Γ^(t)(r) ← Γ^(t)(r) + Δ^(t−1)(c)
12:   for (each summary table group S_g ∈ S) do
13:     for (each cell c in C) do
14:       for (each summary table S_i ∈ S_g) do
15:         Update cursor on S_i to partition p that could cover c
16:         Find record r in p that could cover c
17:         // If p contains such an r, perform allocation
18:         if (r ≠ NULL) then
19:           Δ^(t)(c) ← Δ^(t)(c) + Δ^(t−1)(c)/Γ^(t)(r)
```

6.1 Implementation Details for Block

The complete pseudocode for Block is given in Algorithm 4. The partition size for summary table S_i can be exactly determined during the step where D is sorted into summary table order, as described in Section 4.2.

We assume the imprecise summary tables S_i have been partitioned into a collection of summary table groups $\mathcal{S}$ such that for each group $S_g \in \mathcal{S}$, the sum of the partition sizes for $S_i \in S_g$ is less than $|B|$, the size in pages of memory buffer B. Finding the partitioning of summary tables resulting in the smallest number of summary table groups is an NP-complete problem. This problem can trivially be reduced to the 0-1 Bin Packing problem for which several well-known 2-approximation algorithms exist [8].

The step "update cursor on S_i to partition p that could cover c" (lines 7 and 15) is implemented in a similar fashion to the analogous step in Independent. Following our convention, we omit the costs of sorting D into summary table order and the final cost of writing out the Extended Database D^*, since these are common to all algorithms.

Theorem 7. *Let $|B|$ be the size of the buffer, and $|P|$ is the sum*

of the partition sizes for all imprecise summary tables. Let T be the number of iterations being performed, and $|\mathcal{S}|$ the number of summary table groups. The total number of I/Os performed by the Block algorithm is $3T(|\mathcal{S}||C| + |I|)$ I/Os, where $\lceil \frac{|P|}{|B|} \rceil \leq \mathcal{S} \leq 2\lceil \frac{|P|}{|B|} \rceil$. □

Proof. The smallest possible number of summary table groups $|\mathcal{S}|$ is $\lceil \frac{|P|}{|B|} \rceil$, and the actual value for $|\mathcal{S}|$ returned by the 2-approximation algorithm is at most twice this quantity. For each iteration, the total number of required I/Os per summary table group is given by the following expression $\sum_{i=1}^{|\mathcal{S}|}$[2 scans of C] + [2 scans of each summary table in group i]. Each summary table appears in exactly one summary table group. As explained in the proof for Theorem 6, the two scans of each summary table require 3 I/Os per page. □

7. ADDRESSING THE ITERATIVE ISSUE

Both the Independent and Block Algorithms address the locality problem, and reduce the number of I/O operations required in each iteration. However, for these algorithms, the work performed for an iteration is independent of work for subsequent iterations. Specifically, once a cell or imprecise record is read into memory for an iteration, only work specific to that iteration is performed. Additionally, for both Block and Independent, each subsequent iteration involves the same amount of work as the first iteration of the algorithm.

In this section, we consider improving the Block algorithm to exploit *iterative locality*, allowing the re-use of an I/O operation across several iterations. Once an imprecise record r has been read into memory, we would like to determine the final allocation weights $p_{c,r}$ before r is written back to disk. More generally, we consider the following problem: *Is it possible to partition the allocation graph into subgraphs so that each subgraph can be processed independently for all iterations?* If so, we obtain a significant improvement because the smaller subgraphs which fit in memory can be processed fully without incurring any additional I/O costs for all iterations. The remaining large subgraphs can be handled by reverting to the external Block algorithm described earlier.

To address this problem, we re-examine the Basic Algorithm, listed in Algorithm 1. For the initial iteration of the algorithm, consider a fixed imprecise fact r. Which quantities are *directly* involved in computing the updated $\Delta^{(1)}(c)$ values for cells c in $reg(r)$ (i.e., cells representing possible completions of r could complete to)? In terms of G, using a quantity associated with a node is equivalent to *touching* that node. From line 9, we see $\Gamma^{(0)}(r)$ is computed using $\Delta^{(0)}(c)$ values (i.e., the corresponding nodes in G are touched) for all cells c adjacent to r in G. Similarly, in line 14, the nodes touched to compute $\Delta^{(1)}$(c) are the cell c and imprecise facts r adjacent to c in G. More generally, we have the following:

Theorem 8. *Fix a set of imprecise facts $I' \subseteq I$. Let $C' = \{c \mid (c, r)$ for some $r \in I'\}$ denote the cells that are the neighbors of the facts in I'. Then, the nodes that are touched in an iteration t' in order to compute the values $\Gamma^{(t')}(r)$ for all $r \in I'$ in the first pass belong to $I' \cup C'$. Similarly, for a set of cells C', the nodes that are touched in order to compute the values $\Delta^{(t')}(c)$ for all $c \in C'$ in the second pass belong to C' and the neighbors of C' in G, I''. Thus, the set of nodes touched per iteration is $I' \cup C' \cup I''$.* □

Example 4. In the allocation graph for the sample data in Figure 2, assume we initialize $I' = p9$. Then, $C' = c2, c3$, and $I'' = p7, p12$. □

Intuitively, the set of nodes touched for a particular I' increases in each subsequent iteration, until all nodes reachable from I' are visited. When I' is initialized to a single node r in the graph, this set of nodes is the strongly connected component of the allocation graph G containing r. Since edges in G are undirected, all connected components are strongly connected as well.

Example 5. In the allocation graph for the sample data in Figure 2, there are two connected components: $CC_1 = \{p1, p4, p5, p6, p8, p10, p11, p13, p14\}$ and $CC_2 = \{p2,p3,p7,p9,p12\}$, with "thick" edges in the figure corresponding to edges in CC_2. □

Theorem 9. *Let P be a partitioning of the edges of G into subgraphs $G_1, G_2, \ldots, G_S$ such that each subgraph corresponds to a connected component of G. Then, running the Basic Algorithm with G as the input is equivalent to running the Basic Algorithm on each component $G_1, G_2, \ldots, G_S$ separately across all iterations.* □

Notice the above theorem differs from Theorem 2, which only describes ordering issues *within a single iteration*. This suggests that we should consider partitioning G into the connected components, and the next section presents the Transitive Algorithm based on this idea.

8. TRANSITIVE ALGORITHM

The complete pseudocode for the Transitive algorithm is listed in Algorithm 5. At the highest level, the Transitive Algorithm has two parts. The first identifies connected components in the allocation graph (steps 1 and 2), while the second processes the connected components by performing allocation and creating the EDB entries for facts in the connected component (step 3).

For ease of explanation, we refer to both cells c and imprecise facts r as *tuples*, unless they are treated asymmetrically. We introduce for each tuple t a *connected component id* $ccid$ indicating which connected component t is assigned to. Notice Transitive assigns t a $ccid$ only once, based on information available when t is first considered. However, this $ccid$ may require modification. What is actually a single connected component may be initially identified as several separate components, with tuples in each (incorrectly) assigned different $ccids$. These multiple connected components need to be "merged" by "implicitly" updating the $ccid$ of all tuples to a single value.

This "implicit" merging is accomplished by introducing an auxiliary memory-resident integer array $ccidMap$, where $ccidMap[i]$ corresponds to the "true" $ccid$ of the component assigned $ccid$ i. Our convention is to assign the new "merged" component the smallest $t.ccid$ of any t. The maximum number of entries in $ccidMap$ is the smaller of the following: number of cells in C or number imprecise facts in r, which is comparable to memory-resident data structures used by existing Transitive Closure algorithms [10, 2].

The first step (lines 8 – 19) identifies connected components by assigning a $ccid$ to every tuple t and updating $ccidMap$ appropriately. The processing of cells and imprecise facts for this step is identical to a single pass during one iteration of the Block algorithm. In the second step (lines 21 – 24), all tuples are sorted into *component order* by using the sort key $ccidMap[t.ccid]$. Finally, in step 3 (lines 26 – 34), each connected component is processed, and the EDB entries for tuples in the component are generated. Connected components CC smaller than the buffer B are read into memory, with allocation performed using an in-memory variant of the Block Algorithm, and the EDB entries for tuples in CC are written out. If CC is larger than B, then the external Block algorithm (described in Section 6) is executed, and afterwards, the final

Algorithm 5 Transitive Algorithm

```
1: Method: Transitive Algorithm
2: Input: Allocation Policy A, Cell summary table C, Imprecise Summary Table Groupings S
3: Let |c| be number of cells, |r| number imprecise facts
4: ccidMap ← integer array of size min{|c|, |r|}
5: for (i = 1 to ccidMap.length) do
6:    ccidMap[i] = i
7: // Step 1: Assign ccids to all entries
8: for (each summary table group S ∈ S) do
9:    for (each cell c ∈ C) do
10:      currSet ← {set of r from S_i ∈ S s.t. (c, r) ∈ G} ∪{c}
11:      currCcid ← {set of t.ccid values for t ∈ currSet}
12:      if (currCcid is empty) then
13:         set t.ccid to next available ccid for all t ∈ currSet
14:      else
15:         minCcid ← smallest value for ccidMap[t.ccid] where t ∈ currSet and t.ccid is assigned
16:         for (all t ∈ currSet with unassigned t.ccid) do
17:            t.ccid ← minCcid
18:         for (each cid ∈ currCcid) do
19:            ccidMap[cid] ← minCcid
20: // Step 2: Sort Tuples into Component Order
21: for (i= 1 to ccidMap.length) do
22:    Assign ccidMap[i] = k where k is smallest reachable ccid from ccidMap[i]
23: Let R = C ∪ I
24: Sort tuples t ∈ R by key ccidMap[t.ccid]
25: // Step 3: Process connected components
26: for (each connected component CC) do
27:    if (|CC| < B) then
28:       read CC into memory
29:       evaluate A for tuples in CC
30:       write out EDB entries for CC
31:    else
32:       for (each iteration t) do
33:          perform Block Algorithm on tuples in CC
34:       write out EDB entries for CC
```

EDB entries are generated. Following our convention, we omit the costs of sorting D into summary table order and the final cost of writing out the EDB D^*, since these are common to all algorithms. Additionally, we assume $ccidMap$ remains in memory at all times outside buffer B.

Theorem 10. *Let $|P|$ be the sum of the partition sizes for all summary tables and $|B|$ be the size of the memory buffer (both given in pages). Let T be the number of iterations being performed, and L be the total number of pages containing large components (i.e., components whose size is greater than $|B|$).*

The total number of I/O operations performed by the Transitive Algorithm for all iterations is $2(|\mathcal{S}||C| + |I|) + 5(|C| + |I|) + 3|L|(T + 1)$, where $\lceil \frac{|P|}{|B|} \rceil \leq \mathcal{S} \leq 2\lceil \frac{|P|}{|B|} \rceil$. □

Proof. As with Block, the smallest possible number of summary table groups $|\mathcal{S}|$ is $\lceil \frac{|P|}{|B|} \rceil$, and the actual value for $|\mathcal{S}|$ returned by the 2-approximation algorithm is at most twice this quantity.

For the first step, we are required to scan C for each of the $|\mathcal{S}|$ summary table groups and each imprecise summary table S_i once to assign $ccids$ to all tuples, for a total cost of $2(|\mathcal{S}||C| + |I|)$. The second step requires an external sort of C and all S_i based on ccid value, for a total of $4(|C| + |I|)$ I/Os. The final step involves processing the connected components. Components smaller than $|B|$ are read into memory, all iterations of allocation are evaluated. Only the generated EDB entries are written out, with total cost of $(|C| + |I| - |L|)$ I/Os. In contrast, each large component must be re-sorted again into summary table order (total cost $4|L|$ I/Os for all large components), then external Block algorithm is used on each large component (total cost $3T|L|$ I/Os for all components). Combining these terms together yields the expression in the theorem. □

Observe the only term in the cost formula dependent on the number of iterations T also depends on the total size of the large components $|L|$ as well. Thus, if there are no large connected components in G (i.e., no components with size larger than $|B|$), the number of I/O operations would be *completely independent of the number of iterations*. Since we have established that using the connected components for evaluating the allocation equations is correct, all that remains to be shown is that Transitive correctly identifies these components.

Theorem 11. *The Transitive Algorithm correctly identifies the connected components in the allocation graph G.* □

9. MAINTAINING THE EXTENDED DATABASE

The algorithms presented for creating the Extended Database can be viewed as "end-to-end" algorithms. For a given imprecise fact table D, the entire Extended Database D^* is created by applying the selected allocation policy A to D. Thus, we can view D^* as a *materialized view* over D resulting from this "special" query. An interesting issue becomes efficiently maintaining the EDB view D^* to reflect updates to D. At the highest level, an efficient *view maintenance algorithm for the EDB D^** performs two steps: 1) Identify the entries in EDB D^* whose allocation weight may change due to the update and 2) calculate the updated allocation weight for each of these entries. First, we present the following theorem.

Theorem 12 (Updating the Extended Database)**.** *Let D be a given fact table and D^* be the EDB created by applying allocation policy A to D. Assume D is updated by inserting / deleting / updating a fact r with $reg(r)$ (i.e., region covered by fact r is $reg(r)$).*

Then, the only entries in $f^* \in D^*$ whose allocation weights possibly change correspond to facts f in connected components which overlap $reg(r)$). □

This theorem indicates identifying the entries in D^* that may change as a result of the update to D is a non-trivial task, and why a straightforward application of prior work for materialized view maintenance is not possible. However, this theorem does suggest Step 1 of an EDB view maintenance algorithm can be supported using a spacial index (e.g., R-tree [12]) over the bounding boxes for the connected components in the allocation graph for D (i.e., for each connected component in the allocation graph, compute the bounding box for all its tuples). Step 2 requires efficient access to all facts in D in each connected component (and the corresponding EDB entries). In other words, D needs to be sorted by connected component id (ccid).

These items can be easily obtained from the result of the component identification step of the Transitive Algorithm. After this step completes, D has been sorted into connected component order (i.e., all facts in D in the same connected component are adjacent.) The corresponding EDB entries will be generated in the same "ccid" order, and can be stored in a separate relation. For each connected component CC, we create the bounding box for the connected component, and insert the bounding box for CC into the R-tree, with the bounding box "pointing" to the corresponding facts in D and existing entries in D^*. This step can be "piggybacked" onto the component processing step of the Transitive algorithm.

Given this index, maintaining the EDB D^* is accomplished as follows: Let q correspond to the region for the updated fact in D.

1) Query the R-tree with region q, and find all components whose bounding boxes overlap q. 2) Fetch all facts from D in these overlapped connected components. If many facts in D are in overlapped connected components, then it may be more efficient to scan D entirely. 3) Run Block algorithm over these facts to generate the updated EDB entries, and replace the existing EDB entries with the new entries. 4) Update the R-tree appropriately. If the update to D was an insertion or deletion, then the connected components in the allocation graph may change, and the bounding boxes for overlapped bounding boxes in the R-tree may require updating to reflect this change. This operation is equivalent to several updates to the R-tree. For simplicity the algorithm is described in terms of updating a single fact. The generalization to handling *batch* updates is straightforward.

10. RELATED WORK

[5] provides an extensive list of related work for aggregating imprecise and uncertain data. Although a great deal of recent work has considered uncertain data [16, 9, 7], this work has not considered OLAP-style aggregation queries. A great deal of inspiration for Independent and Block came from the PipeSort and Overlaps algorithms introduced in [1]. Although the proposed algorithms are similar to the respective existing work, there are significant differences: 1) Both existing algorithms only handle precise fact tables, and are used to materialize Group-By views in the OLAP cube. 2) These algorithms only require processing in a single direction, while Independent and Block require iteration in both directions (i.e., from cells to imprecise facts, and imprecise facts to cells). The Transitive Algorithm was inspired by algorithms for computing the Direct Transitive Closure, notably [2, 10]. The main difference between Transitive and existing work is that Transitive exploits optimizations only possible for the class of undirected bipartite graphs, which the allocation graph G always is. Although not explicitly stated in the description, the Transitive Algorithm can easily be modified to handle non-hierarchical data (i.e., dimension values have a general lattice structure instead of a "tree").

Scaling Expectation Maximization [11] to disk resident datasets was introduced in [4]. That work is based on the observation that sufficient statistics for maintaining clusters can easily be held in memory, and the E and M updating steps need only be applied to these statistics as more data records are scanned. Such an approach does not work for EM-based allocation policies, since the size of the required summary statistics that must be held in memory is proportional to the size of the largest connected component in the allocation graph. As our experiments show, we cannot assume this always fits into the memory buffer. The conditions for emergence of a large connected component in general random graphs was studied in [3], but are not directly applicable in our setting since the allocation graphs for an imprecise fact table are not random. The presence or absence of any two edges are not independent events (i.e., all edges in the graph for a particular imprecise fact are either present or absent).

11. EXPERIMENTS

To empirically evaluate the performance of the proposed algorithms, we conducted several experiments using both real and synthetic data. The experiments were carried out on a machine running Windows XP with a single Pentium 2.66 GHz processor, 1GB of RAM, and a single IDE disk. All algorithms were implemented as standalone Java applications.

Since existing data warehouses cannot directly support multidimensional 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 2. Two of the dimensions (SR-AREA and BRAND) have 3 level attributes (including ALL), while the other two (TIME and LOCATION) have 4.

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 2: Dimensions of Real Dataset

Each column of Table 2 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.

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.

The first group of experiments evaluate the performance and scalability of the proposed algorithms, while the second evaluates the efficiency of the proposed maintenance algorithm for the Extended Database based on the R-tree.

11.1 Algorithm Performance

This set of experiments evaluates the performance of the algorithms. All algorithms were implemented as stand-alone Java applications with memory limited to a restricted buffer pool, allowed us to study disk I/O behavior while running experiments small enough to complete in a reasonable amount of time. The important parameter for all of these experiments is the ratio of the input fact table size and the available memory. We set the page size to 4KB, and each tuple was 40 bytes in size.

The first experiment considers the case where the entire fact table fits into memory, and is intended to directly compare the CPU time each algorithm requires for in-memory computation. For these experiments, the buffer was set to 40 MB while the fact table for each data set was approximately 32 MB.

The algorithms were evaluated on two datasets. The first was the real-world Automotive dataset described above. All imprecise facts in the fact table belong to one of 35 imprecise summary tables, and the largest connected component had 7,092 tuples. The second dataset was synthetically generated with the same number of precise and imprecise facts as the Automotive dataset, but imprecise facts were now allowed to take the value ALL for at most two dimensions. For this synthetic data, there were 126 possible imprecise fact tables, and the largest connected component had 167,590

tuples. Both of these items made the synthetic data computationally more challenging than the real data.

Each algorithm was executed on both datasets until the $\Delta(c)$ value for each cell c converged, with different ϵ values used to define convergence (refer to explanation in Section 3.2). It is useful to think of each value of ϵ as corresponding to a number of iterations. For example, in the Automotive data, ϵ values of 0.1, 0.05, 0.01, and 0.005 correspond to 2,3,4 and 6 iterations respectively. In the synthetic data with the large connected component, the corresponding numbers of iterations for these ϵ values were 3,4,6 and 10 iterations respectively. We emphasize one should be careful reading too much into results from a single dataset regarding the required number of iterations. For datasets with more facts and/or dimensions, tens of iterations may be required for the allocation weights to converge for a given ϵ value.

We make two observations regarding the convergence for the $\Delta^{(t)}(c)$ value assigned to cell c: 1) If c is not overlapped by any imprecise facts, then $\Delta^t(c)$ never changes from the initial assigned value (i.e. $\Delta^t(c) = \Delta^{(0)}(c) = \delta(c)$ or all iterations t) and 2) More generally, as the size of the connected component containing c increases, the number of iterations required for $\Delta(c)$ to converge tends to increase as well. Thus, if c is in a small connected component CC, fewer iterations over tuples in CC are required for the $\Delta(c)$ to converge than for larger connected components.

All three algorithms can exploit the first observation as follows: During the first iteration, identify cells c not overlapped by any imprecise facts and ignore these in subsequent iterations. For Transitive alone, a further optimization is possible: For each connected component CC, iterate over entries in CC until $\Delta(c)$ for each cell c converge. This allows the number of iterations to vary from component to component, and only the necessary number of iterations are performed on any given component. If many connected components are small and require few iterations to converge, this significantly reduces the number of allocation equations evaluated by Transitive relative to the other algorithms. Conceptually, Independent and Block must perform the same number of iterations for all cells, which in many cases leads to continued iteration over already converged cells and is wasted effort! *These straightforward optimizations were included in the algorithm implementations.*

The results for the real Automotive dataset and the synthetic dataset are shown in Figure 5a and Figure 5b respectively. The reported running times are wall-clock times. Since the I/O operations are identical in this setting (each fact table is read into memory once and EDB entries written out), relative differences in running time between algorithms can be explained in terms of required in-memory computation for each algorithm.

Independent always does worse than both Block and Transitive, due to the significant CPU processing required to re-sort the cell summary table multiple times for each iteration. The additional overhead of component identification for Transitive results in Block outperforming Transitive for a small number of iterations. However, as the number of iterations increases, Transitive eventually outperforms Block since the savings from detecting early convergence increases. Note the running time of Transitive is very stable as well.

For the second experiment, we studied algorithm performance as the buffer size varied. The same datasets from the above experiment were used here, and have total size of 32 MB with approximately 11 MB of imprecise facts. Thus, the buffer sizes considered ranged from holding all imprecise facts (12 MB) to holding approximately 5% of the imprecise facts (600K).

Figures 5c – e show results from the Automotive dataset for various epsilon values (i.e., number of iterations). Results from corresponding experiments for the synthetic dataset are given in Figures 5f – h. All reported running times are wall-clock times. Since I/O operations dominate running time for these experiments, relative performance of the different algorithms can be explained in terms of the required I/O operations for each algorithm.

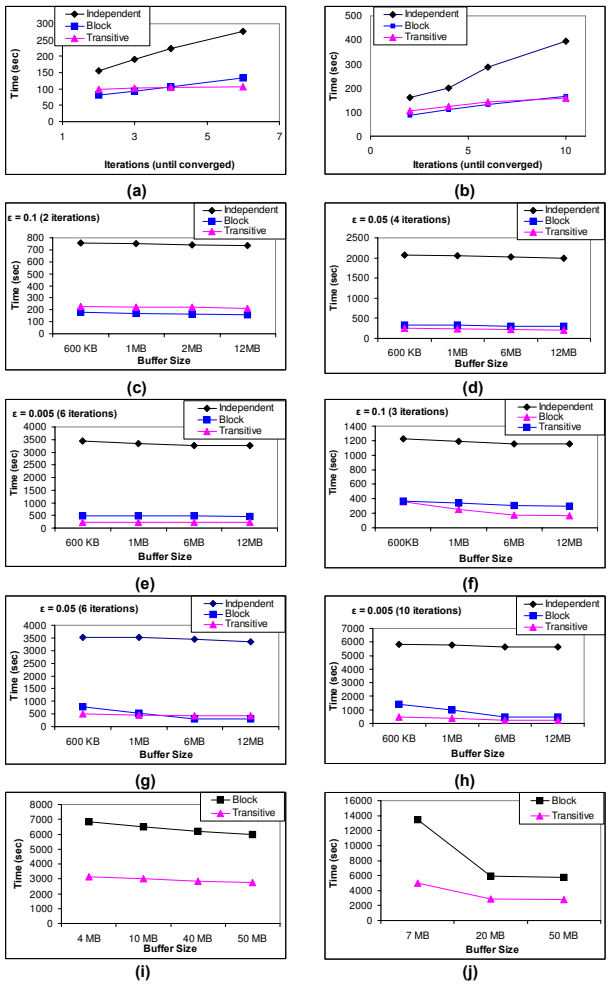

Figure 5: Experimental Results

For the Automotive data, buffer size had negligible impact on running time for all algorithms. The amount of memory Independent requires is independent of buffer size. For Block and Transitive, the total of the partition sizes for the 35 imprecise summary tables was 143 pages, which is smaller than even the 600 KB (150 pages) buffer (i.e., $|\mathcal{S}| = 1$). However, for the synthetic data, the running times for Transitive and Block are impacted by the buffer size. The total partition cost for the 126 imprecise summary tables is 419 pages (1.7 MB). Thus, in terms of the I/O analysis given in Section 6 and 8, for a buffer size of 1 MB, $|\mathcal{S}| = 2$, and for 600 KB, $|\mathcal{S}| = 3$.

From the experiments, Independent performed worse than Block and Transitive, since Independent's I/O cost is dominated by the width of the summary table partial order. Block outperforms Transitive for few iterations. However, as the number of iterations increase, Transitive eventually outperforms Block. For both datasets, the number of I/O's required for Transitive is more stable with respect to the number of iterations than Block. In both datasets, most tuples are in components which fit entirely into the buffer and are read into and written from the buffer once regardless of the number

of iterations.

The third experiment investigates scalability of the algorithms for larger input sizes with proportionately larger memory sizes, and is otherwise similar to the second experiment. Since Block and Transitive clearly dominate Independent, we did not include Independent in this experiment. We created two synthetic datasets, each having 5 million tuples (200 MB) with 30% (1.5 million) imprecise facts. These datasets were otherwise similar to those for the second experiment. The total running times for each dataset for $\epsilon = 0.005$ are shown in Figures 5i and 5j respectively. The relative performance of Block and Transitive is similar to the result from the prior experiment for the same reasons.

11.1.1 Summary of Experiments

Two main conclusions that can be drawn from the performance experiments. First, Independent is a bad idea, due to the extra CPU and I/O overhead of repeated sorting. Second, Transitive provides very stable performance as the number of iterations increases for the price of the component identification step. Although this overhead means Block is more efficient for few iterations, Transitive eventually outperforms Block as the number of iterations increases. We note that all experimental results are consistent with the I/O analysis presented along with the algorithm descriptions in Sections 5, 6, and 8 respectively.

11.2 Extended Database Maintenance

We performed several experiments using the Automotive data to evaluate the performance of the Extended Database maintenance algorithm proposed in Section 9. This data has 283,199 total connected components, of which 205,874 were precise facts not overlapped by any imprecise facts (i.e., a "single" connected component) and 77,325 were connected components with multiple entries. Of these, only 1,152 components had more than 20 entries, 500 components had more than 100 entries, and 93 components had between 1000 and 7092 entries. Thus, most of the connected components in the real Automotive data have few entries.

For the R-tree, we used an open source implementation of a disk-based R-tree available from [13]. The Transitive Algorithm was modified as follows: After the component identification step, the bounding boxes for each identified connected component were generated by scanning the fact D (which was sorted into component order), and bulk loading the R-tree. This process only needs to be performed once.

For these experiments, we generated the following three representative "classes" of update workloads of varying sizes (from between .1% to 10% of the total facts in D): 1) updates to a certain percentage of randomly selected precise facts which are not overlapped by any imprecise facts, 2) randomly selected precise facts (regardless of whether overlapped by imprecise fact or not), and 3) randomly selected facts (whether precise or not). For each query workload, we recorded the ratio of the time taken to update the EDB versus the time taken to recompute the EDB from scratch using Transitive. A value greater than 1 indicates completely rebuilding would have been more efficient.

The results are shown in Figure 6. Updates to the non-overlapped precise facts do not require evaluating any allocation equations, thus the running time is quite stable for all workload sizes. Once overlapping precise facts are selected, performance degrades quickly beyond a few percent regardless of whether the workload contains precise or imprecise facts. The main reason for this is that many entries in the larger connected components are both precise and imprecise facts. Once a fact in a large component is selected, it is irrelevant whether it is precise or not. We note that 2.5% and 5% of the total facts corresponds to roughly 20,000 facts and 40,000 facts respectively. Thus, for a reasonable number of updates, the EDB maintenance algorithm is more efficient than rebuilding the entire EDB from scratch. We note that for these updates the resulting connected component structure did not change, as we only considered updating existing facts.

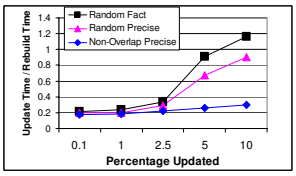

Figure 6: Update Experiment Results

12. CONCLUSION

We proposed several scalable,efficient algorithms for performing allocation. Of the proposed algorithms, the Transitive algorithm is the most intriguing. Its performance is very stable as the number of iterations increases, and the connected components it identifies can be used in an EDB maintenance algorithm we also proposed.

Areas for future work include finding methods for estimating both the number of required iterations to achieve convergence for a given ϵ and size of the largest connected component, and further exploring issues involving EDB maintenance to reflect updates to the fact table.

13. REFERENCES

[1] AGARWAL, S., AGRAWAL, R., DESHPANDE, P., GUPTA, A., NAUGHTON, J. F., RAMAKRISHNAN, R., AND SARAWAGI, S. On the computation of multidimensional aggregates. In *VLDB* (1996).

[2] AGRAWAL, R., DAR, S., AND JAGADISH, H. V. Direct transitive closure algorithms: Design and performance evaluation. In *ACM Transactions on Database Systems* (1990), vol. 15.

[3] BOLLOBÁS, B. *Random Graphs*. Academic Press, London, 1985.

[4] BRADLEY, P., FAYYAD, U., AND REINA, C. Scaling EM (expectation maximization) clustering to large databases. Tech. Rep. MSR-TR-98-35, Microsoft Research Report, August 1998.

[5] BURDICK, D., DESHPANDE, P. M., JAYRAM, T. S., RAMAKRISHNAN, R., AND VAITHYANATHAN, S. Olap over uncertain and imprecise data. In *VLDB* (2005).

[6] BURDICK, D., DESHPANDE, P. M., JAYRAM, T. S., RAMAKRISHNAN, R., AND VAITHYANATHAN, S. Olap over uncertain and imprecise data. In *Submitted to VLDB Journal* (2006).

[7] CHENG, R., KALASHNIKOV, D. V., AND PRABHAKAR, S. Evaluating Probabilistic Queries over Imprecise Data. In *SIGMOD Conference* (2003), pp. 551–562.

[8] CORMAN, T. H., LEIERSON, C. E., AND RIVEST, T. L. *Introduction to Algorithms*. The MIT Press, 2001.

[9] DALVI, N. N., AND SUCIU, D. Efficient Query Evaluation on Probabilistic Databases. In *VLDB* (2004), pp. 864–875.

[10] DAR, S., AND RAMAKRISHNAN, R. A performance study of transitive closure algorithms. In *SIGMOD* (1994), pp. 454–465.

[11] DEMPSTER, A., LAIRD, N., AND RUBIN, D. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society B* (1977).

[12] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD* (1984), pp. 47–57.

[13] HADJIELEFTHERIOU, M. Source code for r-tree implementation. R-tree source code: http://www.dblab.ece.ntua.gr/ mario/rtree/rtree-source.zip.

[14] HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Implementing data cubes efficiently. In *SIGMOD* (1996).

[15] ROSS, K. A., AND SRIVASTAVA, D. Fast computation of sparse datacubes. In *VLDB* (1997).

[16] WIDOM, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR* (2005), pp. 262–276.