RJ10375 (A0603-033) March 24, 2006
Computer Science

IBM Research Report

Efficient Algorithms for Allocation Policies

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

[1]University of Wisconsin
Madison, Wisconsin

[2]IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient Algorithms for Allocation Policies

Doug Burdick‡ Prasad M. Deshpande* T.S. Jayram*
Raghu Ramakrishnan‡ Shivakumar Vaithyanathan*

*IBM Almaden Research Center ‡University of Wisconsin, Madison

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 Data Model (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

Recent work [6] proposed extending the OLAP data model to represent data ambiguity. Specifically, one form of ambiguity that work addressed arose from relaxing the assumption that all dimension attributes in a fact are assigned leaf-level values from the underlying domain hierarchy. Such data was referred to as *imprecise*. In [6], we proposed *allocation* as a mechanism to deal with imprecision. Operationally, allocation is performed by replacing each imprecise fact r in D with a set of precise 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*.

[6] motivated allocation as a mathematically principled method for handling imprecision, and provided a general framework for characterizing the space of allocation policies. However, scalability and performance issues were not explored. The main contribution of this work addresses that is the presentation of scalable algorithms for addressing the following problem:

1. *Given*: Imprecise database D, allocation policy A.
2. *Do*: Materialize Extended Database D^* which results from applying allocation policy A to imprecise database D.

Scalability is an issue because several of the allocation policies presented in [6] were iterative.

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 [6], 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 $h_1 \cap h_2 = \emptyset$. Elements of H are called *imprecise* values. For simplicity, we assume there is a special imprecise value ALL such that $h \subseteq \texttt{ALL}$ for all $h \in H$.

Each element $h \in H$ has a *level*, denoted by $\textsc{level}(h)$, 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 dom(A_i), and (ii) each measure attribute $M_j, j \in 1 \ldots n$, has an associated domain 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 $a_i \in$ dom(A_i) and LEVEL$(a_i) = \ell_i$, for $i \in 1 \ldots k$, and $m_j \in$ 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 $a_i \in$ 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 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. 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). □

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 [5]. 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 [6, 5].

Definition 4 (Allocation Policy and Extended Data Model)**.** Let r be a fact in the fact table. For each cell $c \in$ 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

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

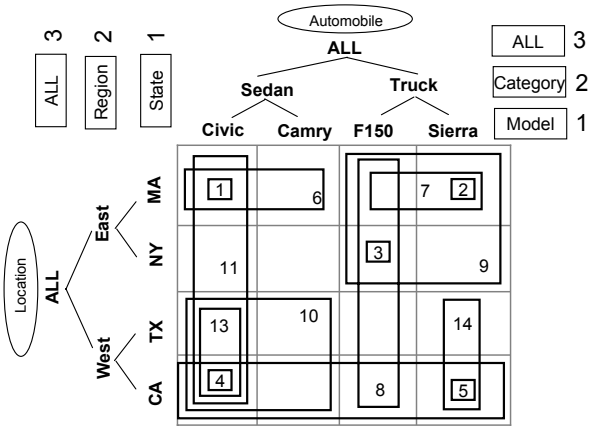

Figure 1: Multidimensional View of the Data

ID	*FactID*	*Loc*	*Auto*	LocL	AutoL	*Sales*	*Weight*
1	p1	MA	Civic	1	1	100	1.0
2	p2	MA	Sierra	1	1	150	1.0
3	p3	NY	F150	1	1	100	1.0
4	p4	CA	Civic	1	1	175	1.0
5	p5	CA	Sierra	1	1	50	1.0
6	p6	MA	Camry	1	2	100	1.0
7	p7	MA	Sierra	1	2	120	1.0
8	p8	CA	Camry	1	3	160	0.5
9	p8	CA	Sierra	1	3	160	0.5
10	p9	MA	Sierra	2	2	190	0.5
11	p9	NY	F150	2	2	190	0.5
12	p10	NY	F150	2	2	200	1.0
13	p11	MA	Civic	3	1	80	0.5
14	p11	CA	Civic	3	1	80	0.5
15	p12	MA	Sierra	3	1	120	0.5
16	p12	CA	Sierra	3	1	120	0.5
17	p13	CA	Civic	2	1	70	1.0
18	p14	CA	Sierra	2	1	90	1.0

Table 2: Extended Database for Sample Data

takes as its input a fact table consisting of imprecise 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*, 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 $\sum p_{c,r} = 1$. By default, each precise fact has a single allocation of 1 for the cell to which it maps. □

Table 2 shows the example of a possible Extended Database that can result from applying an allocation policy to the example data from Table 1.

3.2 Allocation Policy Template

In [5], we demonstrated how the space of allocation policies considered in [6] 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 [5])uses fact count as the allocation quantity. Each fact r is assigned the value 1 (i.e., has a "count" of 1), and each cell c is assigned the "count" of facts "mapped" to 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 [5].

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:

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

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

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 each 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 [6, 5] 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 ϵ ¡ 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 iterations stop. 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 [6, 5], and will not be revisited in this work. However, we will describe the "intuition" behind such iterative allocation policies. Intuitively, 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.

3.3 Allocation Graph

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 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 [6, 5] 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* that describes 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 given 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

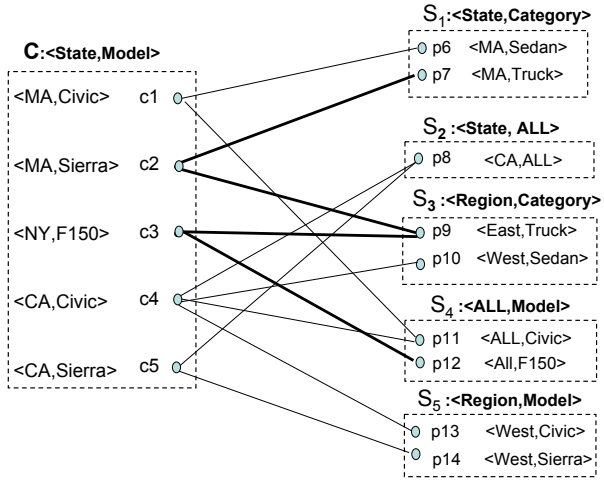

Figure 2: Allocation graph for data in Table 1

and the $\Delta^{(t)}(c)$ equation for cell c. Lines 8 - 10 of the Basic Algorithm correspond to evaluating $\Gamma^{(t)}(r)$ equations for all imprecise facts r. The updates to $\Gamma^{(t)}(r)$ in Line 10 corresponds to evaluating the allocation equation for fact r (generated from Equation 1 of the allocation template). Similarly, Lines 14 - 16 correspond to evaluating $\Delta^{(t)}(c)$ equations for all cells c, with the processing of edges and update in Lines 15 - 16 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 2 passes over the edges of G, and in each pass, each edge of G is processed exactly once. Moreover, the 2 passes cannot in general be replaced by a single pass because the second pass uses values computed for $\Gamma^{(t)}$ in the first pass of the *current* iteration to update the values for $\Delta^{(t)}$.

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:    for (each imprecise fact r) do
 6:       Γ^(t)(r) ← 0
 7:    // Compute t-th step estimate for Γ's
 8:    for (each imprecise fact r) do
 9:       for (each cell c s.t. edge (c, r) ∈ G) do
10:          Γ^(t)(r) ← Γ^(t)(r) + Δ^(t−1)(c)
11:    for (each cell c) do
12:       Δ^(t)(c) ← δ(c)
13:    // Compute t-th step estimate for Δ's
14:    for (each cell c) do
15:       for (each imprecise fact r s.t. edge (c, r) ∈ G) do
16:          Δ^(t)(c) ← Δ^(t)(c) + Δ^(t−1)(c)/Γ^(t)(r)
```

3.4 Scalability of The Basic Algorithm

As presented, the Basic Algorithm has several issues in 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, we observe the nested loops in lines 8 - 10 require access to all cells $c \in C$ in $\mathrm{reg}(r)$ for each imprecise fact r. In general, no single ordering of C exists which provides such locality for all imprecise facts $r \in I$. From the example in Figure 2, notice, that an ordering of C that works for fact $p8$ would not work for $p12$. The nested loops in lines 14 - 16 require access to all imprecise facts $r \in I$ overlapping c for each cell c. Again, there is no single ordering of the imprecise facts I that results in all imprecise facts overlapping a cell being in a contiguous block. We refer to this problem as the *locality issue*.

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, notice each iteration involves two passes over all edges in allocation graph G (i.e., one pass for the nested loops in lines 8 - 10 and a second for the nested loops in lines 14 - 16.) 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. 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.*

4.1 Summary Tables

In order to study appropriate partitions of the imprecise facts for Algorithm 2, it will be helpful to group together imprecise facts according to the levels at which the imprecision occurs. We formalize this notion below.

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 imprecise fact r in P_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 imprecise fact r in P_i) do
13:            // Update Γ^(t)(r)
```

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 identical assignment to their 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 [9].

Definition 8 (Partial Ordering of Summary Tables ($\preceq$))**.** Let $\mathcal{S}$ be the collection of summary tables for D, with the level-vector for summary table S_i denoted as $level(S_i)$. Then, for each $S_i, S_j \in mathcalS$, $S_i \preceq S_j$ iff each element in $level(S_i)$
$leqlevel(S_j)$ and there does not exist any $S_k \in mathcalS$ such that $level(S_i)$
$leqlevel(S_k)$
$leqlevel(S_j)$.

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 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 a pair of level sets associated with that table. For example, the summary table (State,Category) consists of all facts whose level attribute assignment equals $(1, 2)$. Notice the entries in C are *not* precise facts. □

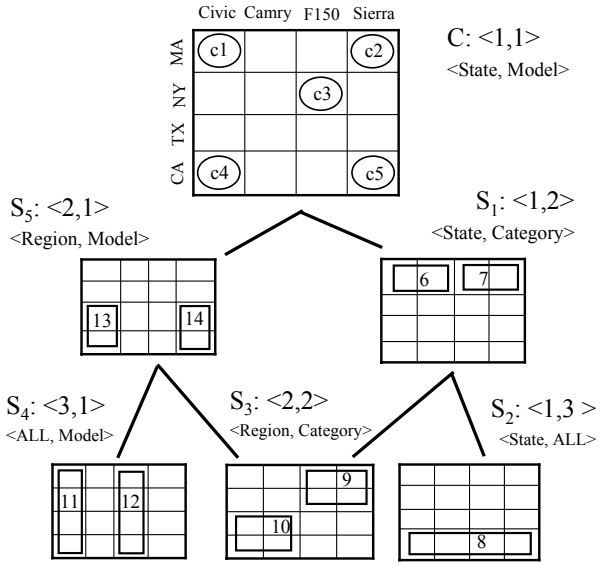

Figure 3: Summary Tables for Example Data (with partial order indicated)

Why are summary tables important in the context of Algorithm 2? 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 this subgraph, it is possible to order 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. The cells adjacent to $p11$ of summary table S_4 are $c1$ and $c4$; however, any contiguous block including these two cells also intersects imprecise fact $p12$. Thus, it appears we have to re-sort C to process S_4. However, if we had enough space available in memory to simultaneously hold all imprecise facts in S_4 whose processing has not been finished, it is still possible to use the present 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 of C into contiguous blocks (i.e., respecting the sort order) is *legal* if for every imprecise fact, all of its neighbors 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. 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 S_i are compared to determine their ordering in S_i. The partition boundaries can only occur between these sorted pair of facts. For each $r \in S_i$, we can easily determine the smallest and largest indexes of entries in C such that edge $(c, r) \in G$, which we denote r.first and $r.last$ respectively. A partition boundary in S_i occurs between entries r_1, r_2 if $r_2.first > r_1.last$. This signifies that all edges have been visited for r_1 before the first edge of r_2 will be visited. This 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 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 $c1, \ldots, c12$.

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 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. A great deal of inspiration for Independent came from the PipeSort algorithm, introduced in [1]. A comparison between the two is provided in Section 9.

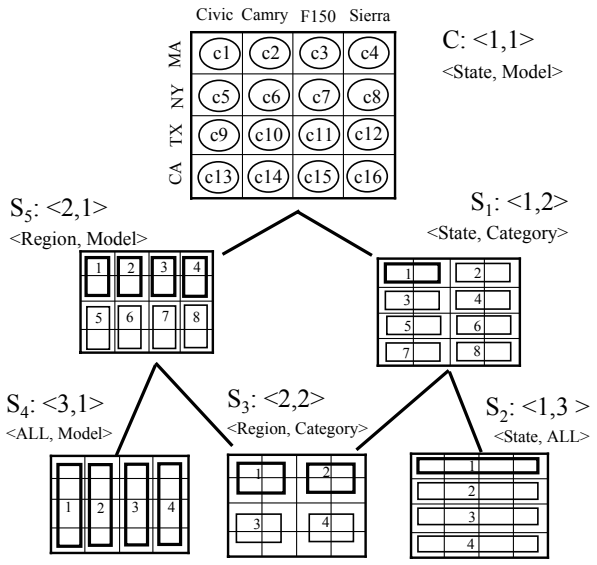

Figure 4: Illustrative Example of Determining Partition Sizes

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 $C, S_1 \preceq S_2 \preceq \cdots \preceq S_k$. There exists a sort order L over all S_i 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 C and the S_i.* □

The proof for Theorem 5 relies on the same observation of the graph structure that Theorem 3 does, namely the degree of each cell node in the allocation graph for each S_i has degree 1. In the pseudo-code below, we refer to the single current fact for each summary table as the *summary table cursor*.

After performing the step where D is sorted into summary table order, we have information about which imprecise summary tables have records in the given fact table D. Thus, we can construct the summary table partial order for D. For a given summary table partial order, the result from [10] can be adapted to provide a lower bound on the number of distinct chains (i.e., summary table groups) in this partial order, which also happens to be the minimum number of sorts required of C. The lower bound is the length of the longest anti-chain in the summary table partial order (i.e., the "width"). The algorithm in [10] for generating the minimal number of chains in the Group-By lattice can be easily adapted to a summary table partial order.

5.2 Independent Details

For each summary table in the chain (including the precise summary table C) we only need enough memory to hold a single fact. Since we consider records in page-sized blocks, we actually perform I/Os for an entire page of records. The pseudo-code for the *Independent* algorithm is given in Figure 3. For ease of presentation, the initialization steps are omitted (since they are identical to those described in the Partitioned Basic Algorithm), and we assume

that D has been sorted into summary table order and summary table partial order information is available.

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

The pseudo-code in Algorithm 3 contains the step "Update cursor on S_i to fact r that could cover c." This step can straightforwardly be implemented by examining the dimension attribute values of c and r, and the details are implementation specific. The correctness of this algorithm was established by Theorem 5.

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 ∈ S) do
4:     Sort C and summary-tables in S into sort-order L
5:     // Compute t-th step estimate for Γ
6:     for (each cell c) do
7:       for (each summary table S_i ∈ S) do
8:         Update cursor on S_i to fact 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 ∈ S) do
13:    for (each cell c) do
14:      for (each summary table S_i ∈ S) do
15:        Update cursor on S_i to fact 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 $|I|, |C|$ be the number of pages for imprecise facts $I \in D$ and the cell summary table C respectively. Let W be the length of the longest anti-chain in the summary table partial order, and T be the number of iterations. The Independent Algorithm in the worst case requires $7WT|C| + 7T(|I|)$ I/Os.*

Proof. We make the standard assumption that external sort requires two passes over the relation, with each page requiring a read and write I/O. Each summary table group is sorted into the corresponding sort-order of L. Then, two passes are required over each summary table in the group and the cell-level summary table 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. 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-level summary table C is likely to be prohibitive. In the general case, the number of entries in C will be much larger than the total number of size of the other summary tables. If 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 entries of S_i for which we have seen at least one fact in C and may see at least one more fact in C. 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 fact for each imprecise summary table S_i and the precise summary table 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 fact to a contiguous block of records, which we called the *partition of* S_i. Only a single partition of S_i needs to be held in memory for a summary table as we scan C.

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 ∈ S) do
5:     for (each precise fact c in D) do
6:       for (each summary table S_i ∈ S) do
7:         Update cursor on S_i to partition p that could cover c
8:         Find 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 ∈ S) do
13:    for (each precise fact c in D) do
14:      for (each summary table S_i ∈ S) do
15:        Update cursor on S_i to partition p that could cover c
16:        Find 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 pseudo-code for Block is given in Algorithm 4. The upper bound on a partition size for each summary table S_i can be exactly determined during the step where D is sorted into Summary Table order. The step "Update cursor on S_i to partition p that could cover c" 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 sum of the partition sizes for all summary tables and $|M|$ the size of the memory buffer (both given in pages). Let T be the number of iterations being performed. Let $W' = \lceil \frac{|B|}{|M|} \rceil$ be the number of summary table groups. The total number of I/Os performed by the Block algorithm is between $3W'T|C| + 3T(|I|)$ I/Os and $2[3W'T|C| + 3T(|I|)]$.*

Proof. Finding the smallest value of W is an NP-complete prob-

lem, and there exists a trivial reduction of the problem to the 0-1 Bin Packing problem for which several well-known 2-approximation algorithms exist [7]. For each iteration, the total number of required I/Os per summary table group is given by the following expression $\sum_{i=1}^{W'}$[2 scans of C] + [2 scans of each summary table in group i], which when expanded gives the expression in the theorem since each summary table appears in exactly one summary table group. □

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 fact are read into memory for an iteration, only work specific to that iteration is performed. Additionally, for both Block and Independent, the amount of work performed for each iteration is the same as the amount of work performed in the first iteration of the algorithm.

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

To address this problem, let us re-examine the Basic Algorithm, and first consider a simpler question: For a fixed imprecise fact r in the allocation graph G, and a fixed iteration t', what quantities are required to compute $\Gamma^{(t')}(r)$ in the first pass? From Line 10, the answer is simple—we need all the values $\Delta^{(t'-1)}(c)$ for all cells c that are neighbors of r in G. Since $\Gamma^{(t')}(r)$ is updated incrementally, this effectively means that for each iteration, the only nodes that are *touched* (i.e., value in that node is used) for computing $\Gamma^{(t')}(r)$ are just r and its neighboring cells in G. Similarly, the nodes that are touched for computing $\Delta^{(t')}(c)$ are the cell c and its neighboring imprecise facts 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 iteration, until all nodes reachable from I' are visited. When I' is initialized to a single node r in the graph, this 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 2 connected components: $CC_1 = \{p1, p4, p5, p6, p10, p11, p12, p13, p14\}$ and $CC_2 = \{p2, p3, p7, p8, p9\}$. In the example allocation graph in Figure 2, the "thick" edges correspond 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* an 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

At the highest level, the Transitive Algorithm has two parts. The first part identifies the connected components in the allocation graph. The second part processes the connected components to perform allocation and create the EDB entries for the facts in the connected component.

For ease of explanation, we will refer to both cells and imprecise facts as *tuples*, unless they are treated asymmetrically. Thus, the set of all tuples is the union of all cells c and imprecise facts r. We introduce for each tuple t a *connected component id* $ccid$ indicating which connected component t is assigned to. The algorithm assigns $ccid$ only once. However, during algorithm execution, multiple connected components may need to be "merged" (i.e., a single component was identified as multiple separate components, with each assigned a unique $ccid$). We now need to update the $ccid$ of all tuples in the merged component. This is accomplished "implicitly" by introducing an auxiliary memory-resident integer array $ccidMap$, where $ccidMap[i]$ corresponds to the "true" $ccid$ of the component assigned $ccid$ i. The size of $ccidMap$ is the smaller of the number of cells or number of imprecise facts. We note the size of $ccidMap$ is comparable to memory-resident data structures used by existing Transitive Closure algorithms [8, 2]. Our convention is to assign the new "merged" component the smallest $t.ccid$ of any tuple t identified to be in the component.

The Transitive Algorithm has three steps. In the first step the connected components are identified. In other words, a $ccid$ is assigned to every tuple t, and $ccidMap$ is appropriately updated. The processing of cells and imprecise facts for this step is identical to a single pass of the Block algorithm. For the second step, all tuples t are sorted into component order by using the sort key $ccidMap[t.ccid]$. Finally, in step 3, each connected component is processed, and the Extended Database entries for the tuples in the component are generated. Connected components CC which are 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 EDB entries are generated.

The complete pseudo-code for the Transitive Algorithm is given in Figure 5.

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 10. *Let $|B|$ be the sum of the partition sizes for all summary tables and $|M|$ be the size of the memory buffer (both given in*

Algorithm 5 Transitive Algorithm

Method: Transitive Algorithm
Input: Allocation Policy A, Cell-level summary table C, Imprecise Summary Table Groupings $\mathcal{S}$
Let $|c|$ be number of cells, $|r|$ number imprecise facts
$ccidMap \leftarrow$ integer array of size min$\{|c|, |r|\}$
for (i = 1 to $ccidMap.length$) **do**
  $ccidMap[i] = i$
// Step 1: Assign ccids to all entries
for (each summary table group $S \in \mathcal{S}$) **do**
  for (each cell $c \in C$) **do**
    $currSet \leftarrow$ {set of r from $S_i \in S$ s.t. $(c, r) \in G$} $\cup\{c\}$
    $currCcid \leftarrow$ {set of t.ccid values for $t \in$ currSet}
    if ($currCcid$ is empty) **then**
      set t.ccid to next available ccid for all $t \in currSet$
    else
      $minCcid \leftarrow$ smallest value for currMap[t.ccid] where $t \in currSet$ and t.ccid is assigned
      for (all $t \in$ currSet with unassigned $t.ccid$) **do**
        $t.ccid \leftarrow minCcid$
      for (each $cid \in currCcid$) **do**
        currMap[cid] $\leftarrow minCcid$
// Step 2: Sort Tuples into Component Order
for (i= 1 to currMap.length) **do**
  Assign currMap[i] = k where k is smallest reachable ccid from currMap[i]
Let $R = C \cup I$
Sort tuples $t \in R$ by key currMap[t.ccid]
// Step 3: Process connected components
for (each connected component CC) **do**
  if ($|CC| < B$) **then**
    read CC into memory
    evaluate A for tuples in CC
    write out EDB entries for CC
  else
    for (each iteration t) **do**
      perform Block Algorithm on tuples in CC
    write out EDB entries for CC

pages). Let $W = \lceil \frac{|B|}{|M|} \rceil$. 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 $|B|$). Assume that $ccidMap$ remains in memory at all times, but is outside of buffer M.

The total number of I/O operations performed by the Transitive Algorithm for all iterations is between $(2W + 5)(|C|) + 7|I| + (3T + 2)|L|$ *and* $(4W + 5)(|C|) + 7|I| + (3T + 2)|L|$ □

Proof. As with Block, finding the smallest value of W is an NP-complete problem with a well-known 2-approximation. For the first step, we are required to scan C for each of the W summary table groups and each summary table once to assign the ccids to all tuples, for a total cost of $2W|C|+2|I|$. The second step requires sorting C and all summary tables based on ccid value, and we assume external sort requires 2 passes, 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 and have all iterations of allocation evaluated. Only the Extended Database entries are required to be written out, thus the total cost is $(|C| + |I| - |L|)$. Large components must be processed using Block for each iteration, for a total for all large components of $(3T + 3)|L|$ I/Os. First, each large component must be re-sorted again into summary table order, for a total cost of $4|L|$ I/Os. The Block algorithm requires $(3T - 1)$ I/Os, with the final write of the component replaced by only writing out the EDB entries. Combining these terms together yields the above expression. □

Observe the only term in the cost formula dependent on the number of iterations T 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 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. RELATED WORK

Independent uses the structure between summary tables in a manner similar to how PipeSort uses the structure between Group-By views to materialize the cube. In PipeSort, the idea was a single entry in each Group-By view in the pipe needed to be held in memory. The concept of a Group-By view is identical to our notion of a summary table. While PipeSort will generate all entries in a Group-By view with corresponding precise records, Independent is interested only in the summary table entries that have facts in the given instance D.

The other major difference is how the "pipes", or "chains", are used. A reasonable implementation of PipeSort would explicitly traverse the chains in order. For Independent, the chains are "implicit", in the sense that a reasonable implementation would consider the summary tables in the chain in any order (i.e., allocation equations are evaluated using only allocation statistics from the precise summary table C and one of the imprecise summary tables in the chain). Thus, the chain is used only to describe the grouping of summary tables for processing. The reason for this difference is that Independent must traverse the chains in both the "up" direction from C to the end, and "down" direction. Also, for Independent, the precise summary table C is part of every chain, or summary table group. The reason is that the "down" direction involves actually modifying the allocation statistics for each precise fact in C.

The Block algorithm is similar in spirit to the Overlaps algorithm for materializing the OLAP cube, presented in [1]. The Overlaps algorithm was based on re-using the same sort order to compute several Group-By views. There are two main differences between Overlaps and Block, which are analogous to the differences between PipeSort and Independent. First, since Overlaps handled precise facts, every entry containing a fact in a Group-By view was created (similar to PipeSort). For Block, we are only interested in entries in the Group-By view (i.e., facts in a summary table) corresponding to an imprecise fact in D. In practice, this provides two distinct advantages. First, the latter is significantly smaller than the number of entries in the Group-By view. Second, the *exact partition size* for each summary table is available after D has been sorted into summary table order. As presented in [1], Overlaps could either place an analytical upper bound on partition size based on the dimension hierarchies (and dimension ordering) or could use a tighter heuristical estimate based on the statistics of the data instance. Having the exact size of each summary table partition available makes such estimates unnecessary for the proposed Block algorithm.

Second, the Block algorithm requires processing summary tables in the "up" direction and the "down" direction. For this reason, a reasonable implementation of Block would disregard the structure between summary tables. The reason is that the "down" direction involves actually modifying the allocation statistics for each precise fact c, and it would be easier to directly process the entries in C for each S_i directly.

The Transitive Algorithm was inspired by algorithms to compute the Direct Transitive Algorithm [2], and a comparison between these algorithms and Transitive is made in Section 9.

[8]
[2]
[4]
[3]

10. EXPERIMENTS

To empirically evaluate the performance of the proposed algorithms, we conducted several experiments using both real and synthetic data. All algorithms were implemented in Java, and the experiments were carried out on a machine with a single Pentium 2.66 MHz processor, 1GB of RAM, and a single IDE disk.

Since existing data warehouses cannot directly support imprecise data, there are not many "real-world" examples of multidimensional imprecise data with hierarchical dimensions. However, we were able to obtain one such real-world dataset from an automotive manufacturer. The fact table contains 797,570 facts, of which 557,255 facts were precise and 240,315 were imprecise (i.e., 30.1% 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 MODEL) have 3 attributes (including ALL), while the other two (TIME and LOCATION) have 4. Each column of Table 3 lists the characteristics of each attribute for that dimension. Next to each attribute name are two numbers. The first lists the number of distinct values the attribute can take, and the second gives the percentage of facts in the fact table taking a value from that attribute for the particular dimension. For example, for the SR-AREA dimension, 92% of the facts take a value from Sub-Area attribute, while 8% take a value from the Area attribute.

Although a significant number of facts were imprecise, most imprecise facts has imprecision which was fairly moderate. Most imprecise facts had imprecise values for a single dimension, and most imprecise values were from attributes one level above the lowest leaf level in the dimension. Of the imprecise facts, approximately

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

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. No imprecise fact had the attribute value ALL.

An alternative characterization of the imprecision in a given fact table involves the relative number of facts in each summary table. Recall the summary table partial order, introduced in Section 4. Consider two summary tables S_1, S_2. If $S_1 \preceq S_2$, then facts in S_2 are more imprecise then facts in S_1, since each fact in S_2 corresponds to a larger region.

Since we are interested in the performance of our proposed algorithms in real-world scenarios, many of the parameter settings for the synthetic data we used in the experiments attempt to mimic this real-world data. To be concrete, the synthetic data used the same 4 dimensions as the real-world data. The general 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 dimensions. The

The first groups of experiments attempts to determine which factors affect the size of the connected components in the allocation graph for a imprecise fact table D. The second groups of experiments evaluate the performance and scalability of the proposed algorithms. The third group of experiments evaluates the efficiency of the proposed maintenance algorithm for the Extended Database using the R-tree.

10.1 Connected Component Size

10.2 Algorithm Performance

This set of experiments evaluates the performance of the algorithms. ll algorithms were implemented to use a buffer pool, thus we could control the memory available to the algorithms. This allowed us to study disk I/O behavior, while running experiments small enough to complete in a reasonable amount of time. We set the page size to 4KB. Each tuple is 40 Bytes, and each page holds 90 tuples.

Experiment 1: Everything fits into memory

For the first experiment, we considered the case where the buffer is larger than the entire dataset, and evaluted the algorithms on two datasets. The first dataset was the Automotive dataset. The second was a synthetically generated dataset with the same number of precise and imprecise facts as the Automotive dataset, but generated to contain a connected component of 200,000 facts.

For this experiment, we set the buffer size to 40 MB. We ran each algorithm on 2 datasets until the values for all cells c converged for different values of epsilon. This is a commonly technique for numerical EM algorithms. Each value of epsilon corresponds to a number of iterations. For example, in the Automotive data, 2,3,4 and 6 iterations corresponded to epsilons of 0.1, 0.05, 0.01, and 0.005 respectively. In the synthetic data with the large connected component, 3,4,6 and 10 iterations corresponded to epsilons of 0.1, 0.05, 0.01, and 0.005 Again, one should be careful reading too much into these single dataset. In many practical settings, tens of

iterations may be required to achieve the desired epsilon.

However, we observed the values for some cells converged in fewer iterations than for other cells. Independent and Block can exploit this observation as follows: During the first iteration, identify cells c not overlapped by any imprecise facts. Such cells can be ignored in subsequent iterations. The Transitive algorithm can make the further optimization to vary iterations for different components; each component is iterated on until all cells in the component converge. This observation can significantly reduce the number of allocation equations that Transitive must evaluate relative to the other two algorithms.

Figure ?? shows the results for running the experiment on the Automotive dataset. Independent does much worse than Block and Transitive. Even though Independent and Block evaluate the same number of allocation equations, Independent requires more processing to re-sort the cell summary table multiple times for each iteration. Block outperforms Transitive for a small number of iterations, since Transitive has the additional overhead of component identification (Steps 1 and 2 of the Transitive Algorithm). However, for many iterations, the savings from reducing the number of allocation equations evaluated by Transitive relative to Block dominates the extra overhead.

Figure 5b shows the results for the synthetic dataset with the large connected component. Independent still does worse for reasons given above. Due to the large conencted component, the optimization for Transitive described above does not provide enough of a savings to overcome the extra overhead of component identification relative to Block.

Experiment 2: Everything does not fit into memory.

We also ran several experiments on datasets with

10.3

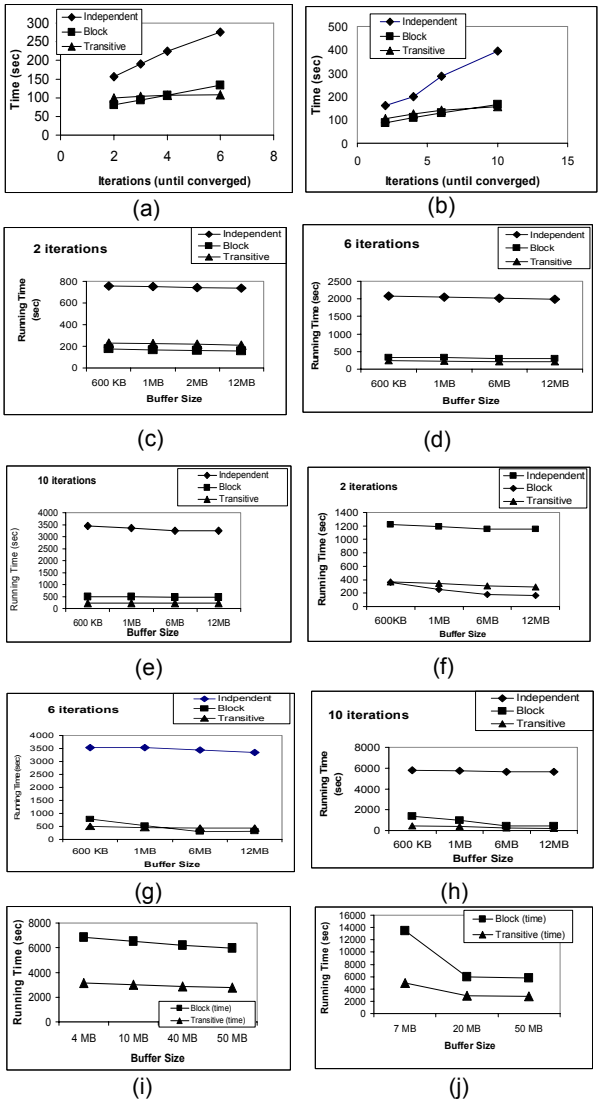

Figure 5: Experimental Results

11. 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), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann, pp. 506–521.

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

[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, 1998.

[5] BURDICK, D., DESHPANDE, P. M., JAYRAM, T. S., RAMAKRISHNAN, R., AND VAITHYANATHAN, S. OLAP Over Uncertain and Imprecise Data. In *Submitted to the VLDB Journal*.

[6] BURDICK, D., DESHPANDE, P. M., JAYRAM, T. S., RAMAKRISHNAN, R., AND VAITHYANATHAN, S. OLAP Over Uncertain and Imprecise Data. In *VLDB* (2005).

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

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

[9] HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Implementing Data Cubes Efficiently. In *SIGMOD* (1996).

[10] ROSS, K. A., AND SRIVASTAVA, D. Fast Computation of Sparse Datacubes. In *VLDB 1997*, pp. 116–125.