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

1. INTRODUCTION

Recent work [2] 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*. *Allocation* was proposed by [2] as a mechanism to deal with imprecision. Intuitively, an allocation policy assigns a weighted portion of an imprecise record to each cell in the region covered by the imprecise record. [2] motivated allocation policies as a mathematically principled method for handling imprecision, and detailed the properties of several allocation policies. The result of applying an allocation policy to an imprecise database D is referred to as an *extended data model*.

1.1 Problem statement

This work presents scalable algorithms for addressing the following problem:

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

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 [2], 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 $h \subseteq$ 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 $\langle A_1, A_2, \ldots, A_k; L_1, L_2, \ldots, L_k; M_1, M_2, \ldots, M_n \rangle$ where (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 (ii) 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 $\langle a_1, a_2, \ldots, a_k; \ell_1, \ell_2, \ldots; m_1, m_2, \ldots, m_n \rangle$ 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

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	Sierra	3	1	120
p13	West	Civic	2	1	70
p14	West	Sierra	2	1	90

Table 1: Sample data

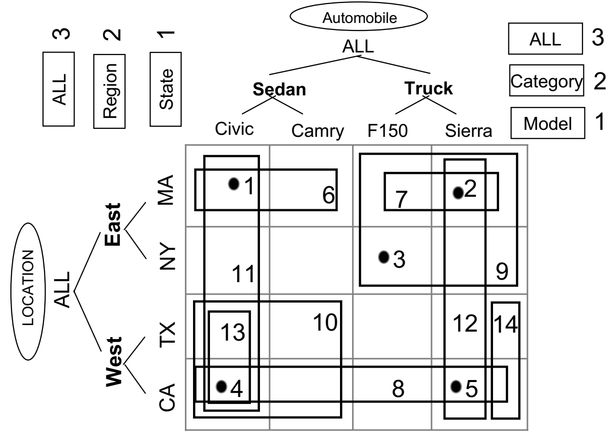

Figure 1: Multidimensional View of the Data

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 *Loc-Level* and *Auto-Level*, respectively.

Precise facts, p1–p5 in Table 1, have leaf nodes assigned to both dimension attributes and are represented as "dots" 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)`. □

Given a fact table, it will be helpful to group together imprecise facts according to the levels at which the imprecision occurs. This notion will be extensively used by our algorithms and is formalized below.

Definition 4 (Summary Tables)**.** Let D be a fact table. The *level vector* of a fact is the assignment of values to its level attributes. Partition the facts in D by grouping together facts in D that have identical level vectors. We refer to each such grouping of the facts as a *summary table*. Note that each summary table is associated with a distinct level vector. The summary table for precise records is referred to as the *cell summary table* and all other summary tables as *imprecise summary tables*.

We define a partial order $\preceq$ on the set of summary tables of D as follows: $S_1 \preceq S_2$ if and only if the level vector corresponding to summary table S_1 is component-wise less than or equal to the level vector corresponding to summary table S_2. □

Intuitively, the summary tables are "logical" groupings which are similar to the result of performing a Group-By query on the level vectors of the facts in D and the summary table relationship is identical to the one defined for Group-By views in [3]. It can be seen that the cell summary table is the *minimum* element of this partial order, and that every chain corresponds to increasing the imprecision along the hierarchy levels of one or more dimensions.

Example 2. Consider the sample data in Table 1. Assume we consider the dimensions in the sample data in the order (*Location*, *Automobile*). The level-vector for facts p1-p5 is (1,1). For p6, the level-vector is (1,2). For this data set, there are 6 summary tables (the precise summary table $C = S_0$ and 5 imprecise ones S_1 - S_5), as indicated in Figure 2. The multidimensional representation for each summary table is shown. Each summary table is labeled by a pair of level sets determined by the level vector associated with that table. For example, the summary table (State,Category) consists of all facts whose level vector equals $(1, 2)$. Observe that $(\text{State}, \text{Category}) \preceq (\text{State}, \text{ALL})$. In the figure, we draw an edge from table S_i to table S_j (by convention, the direction is from top to bottom) if $S_i \preceq S_j$ and there is no table S_k such that $S_i \preceq S_k \preceq S_j$. Thus, there is an edge from $(\text{State}, \text{Category})$ to $(\text{State}, \text{ALL})$. Moreover, $(\text{State}, \text{Model}) \preceq (\text{State}, \text{ALL})$ because there is a path from the first summary table to the second. On the other hand, the tables (State,ALL) and (Region,Model) are incomparable under the partial order. □

Since each summary table is associated with a unique level vector, it is possible to sort the input data D so that all facts in each summary table are contiguous (i.e., facts in summary table 1 are followed by facts in summary table 2, etc). The sorting key is formed by the concatenation of the level-vector and the dimension-vector.

Example 3. Consider fact p1 from the example. The level and dimension vector for p1 are `(1,1)` and `(MA,Civic)` respectively. When D is sorted into summary table order, the sorting key used for fact p1 would be `(1,1,MA,Civic)`. □

3. ALLOCATION POLICIES

Enabling drill-down to the leaf-nodes in the presence of imprecision requires some manipulation. One simple alternative is simply to ignore the imprecise facts; however, this may result in loss of accuracy of the aggregated information. To overcome this in [2] the

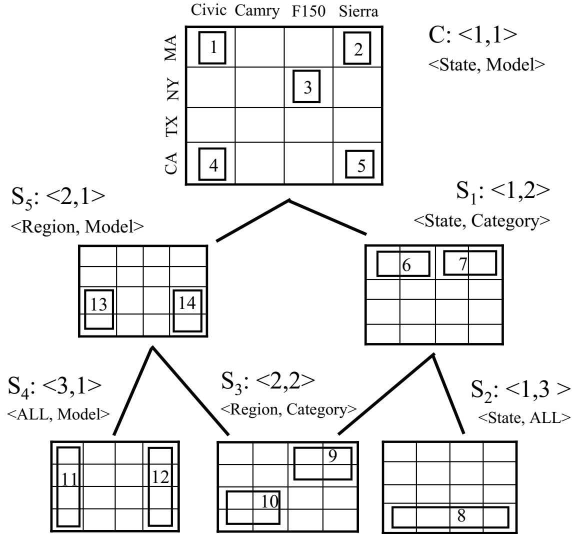

Figure 2: Summary Tables

authors introduced *allocation* as a mechanism to replace imprecise facts by possible completions to precise facts. For completeness we rewrite the following definition from [2].

Definition 5 (Allocation)**.** Every imprecise fact r is replaced by a set of precise facts, each of which maps to a distinct cell in $\text{reg}(r)$, along with weights $p_{c,r} > 0$ that sum to 1. $p_{c,r}$ is called the *allocation* of fact r to cell c.

Any procedure for assigning $p_{c,r}$ values is referred to as an *allocation policy*. The result of applying an allocation policy to a fact table D is an allocated database D', which we refer to as the *Extended Data Model*(EDM). The schema of the EDM contains all of the columns from D plus additional columns to keep track of cells having strictly positive allocation weights. Table 2 shows the example of a possible EDM that can be the result of an allocation policy to the example data from Table 1.

An interesting characterization of the space of allocation policies, largely motivated by the quality of aggregations, is provided in []. Unfortunately, this characterization is not conducive to performance and scalability analysis of the different allocation policies. With this in mind we will provide a bipartite graph representation along with a generalized algorithm that captures all the allocation policies discussed in [2].

4. ARCHITECTURE FOR ALLOCATION

Let I denote the set of imprecise facts in the given data set. These are the facts for which the allocations need to be determined. Let C denote a set of cells representing all the possible completions of facts in I, appropriately determined by the allocation policy. For example, most of the allocation policies in [2] choose C to equal the cells of the precise facts i.e. the cells covered by the cell summary table. Define a bipartite graph G as follows. The vertices on the left correspond to the cells in C, and the vertices on the right correspond to facts in I. There is an edge (c, r) in G if and only if $c \in \text{reg}(r)$. For the data in Table 1, letting C equal the cells of the precise facts results in graph as shown in Figure 3.

The graph described above is the skeleton of our framework. We now describe the dataflow of the allocation policies that use the

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	CA	Camry	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 Data Model for Sample Data

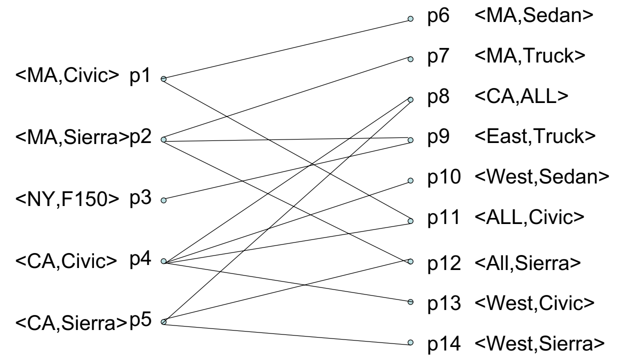

Figure 3: Bipartite graph between cells on the left and imprecise facts on the right. Here the cells are associated with the precise facts of the data

skeleton G. To that end, we associate certain quantities with each node in the graph G which are updated in an iterative manner. Formally, let $\Delta(c)$ denote the quantity associated with every cell c, and let $\Gamma(r)$ denote the quantity associated with every fact r. The values for these quantities at the end of the t-th step of the iteration will be denoted by $\Delta^{(t)}(c)$ and $\Gamma^{(t)}(r)$, respectively. The following pseudocode is the template used by all of the allocation policies to update these quantities.

Algorithm 1 Basic Algorithm

```
1:  for (each cell c) do
2:     Initialize Δ^(0)(c)
3:  for (each iteration) do
4:     for (each imprecise fact r) do
5:        Γ^(t)(r) ← 0
6:     for (each cell c) do
7:        for (each imprecise fact r such that (c, r) is an edge) do
8:           Γ^(t)(r) ← Γ^(t)(r) + Δ^(t−1)(c)
9:     for (each cell c) do
10:       Initialize Δ^(t)(c)
11:    for (each cell c) do
12:       for (each imprecise fact r such that (c, r) is an edge) do
13:          Δ^(t)(c) ← Δ^(t)(c) + h(r, Δ^(t−1)(c), Γ^(t)(r))
```

Note that the updates to $\Gamma^{(t)}$ and $\Delta^{(t)}$ in Line 8 and Line 13, respectively, are constrained to use the edges of G. Second, in order to instantiate this template, we need to specify the function h that is used in Line 8; since r is part of the argument of h, this allows measure and the dimension attributes of r to be used. Third, the operator binary sum in Line 13 can be generalized and yet ensure that the final value of both of $\Gamma^{(t)}(r)$ or $\Delta^{(t)}(c)$ is the same regardless of the order in which we process the updates. This is shown in the following theorem. Moreover, we also show how the allocations can be recovered from these two quantities.

Theorem 1. *Suppose the updates to $\Delta^{(t)}(c)$ are computed using a operator that is commutative, associative, and which has an identity element (this is also known as a commutative monoid). Then, the final values of $\Delta^{(t)}(c)$ and $\Gamma^{(t)}(r)$ will be the same regardless of the order in which we process the updates.*

Moreover, the equations $p_{c,r} = \Delta^{(t-1)}(c)/\Gamma^{(t)}(r)$, for every c and r such that (c, r) is an edge of G define a proper set of allocations.

Example 4. In [2], an EM-based iterative allocation policy was proposed in which the equations for $\Delta(c)$ were defined so as to ensure that the interactions between imprecise facts are taken into account in determining the allocations (see [2] for details). To define this, let C denote the cells corresponding to the precise facts in the data. For $c \in C$, let $\delta(c)$ denote the number of precise facts that map to c, a quantity that will remain fixed throughout the iterations. The equations were defined as follows:

$$\Delta^{(t)}(c) = \delta(c) + \sum_{r:c\in \mathrm{reg}(r)} p_{c,r}^{(t-1)} \qquad (1)$$

$$p_{c,r}^{(t)} = \frac{\Delta^{(t)}(c)}{\sum_{c':c'\in \mathrm{reg}(r)} \Delta^{(t)}(c')} \qquad (2)$$

We now rewrite Equations 1 and 2 so that it corresponds to our basic template. Let $\Gamma^{(t)}(r)$ denote the summation in the denominator of the fractional expression in Equation 2. We rewrite these two equations as follows:

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

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

It is an easy exercise to verify that the above equations are equivalent to Equations 1 and 2.

4.1 Problems that need addressing

Clearly, this algorithm is not feasible for disk-resident fact tables. Each iteration of a loop of the form "for each imprecise fact" or "for each precise fact" involves a complete scan of D. It is not obvious if there is a "good" ordering of the facts in D that can be used for evaluating both loops in an I/O efficient fashion. (i.e., the ordering of D allows all iterations of the for loops to be evaluated in a single scan of D.)

The loop "for each imprecise fact" requires all of the precise facts overlapping a precise fact be near each other in the ordering of D. Similarly, the second for loop requires that every imprecise fact r covered by a precise fact c be near each other. By similar reasoning, a straightforward application of indices does not work, since any index of D would be unclustered with respect to one of the required types of locality (i.e., an index on D can be constructed to access imprecise records).

We refer to the above problem as the *locality issue*. A second orthogonal issue is the *iteration issue*. Even if there existed a "good" ordering of D that supported efficient evaluation of the basic algorithm, D would need to be completely scanned for each iteration. This issue is significant in practice, since all but the smallest fact tables will be larger than main memory and a non-trivial number of iterations will likely be required before the assigned allocation weights converge.

The proposed Independent and Block algorithms directly address the locality issue. The Transitive algorithm directly addresses both the locality and iteration issues.

5. INDEPENDENT ALGORITHM

In this section we describe the Independent algorithm.

5.1 Summary Table Structure

The structure between Group-by views noted in Section 2 was exploited by the PipeSort algorithm, introduced in [1]. Consider a path through the summary table lattice, containing in order summary tables $S_1 \preceq S_2 \preceq \cdots \preceq S_k$. It is possible to define a sort-order for all facts in these summary tables so that all facts in S_1 contained in the region for the first fact in S_2 are a contiguous block, followed by a block of all facts in S_1 contained in the region for the second fact in S_2, and so on. This property is recursive along the path. For example, the facts in S_1 covered by a fact in S_3 will be a contiguous block as well.

This observation was used by the PipeSort algorithm to limit the amount of memory needed to materialize the Group-By views in each path (i.e., pipe) through the lattice. The idea was a single entry in each Group-By view in the pipe needed to be held in memory. It is useful to think of this current entry as a *cursor* on the Group-By view. As the precise records were scanned in order, they would be "piped" to the current entry for each group-by view, and the current entry would be updated appropriately. For a Group-By view, when the current precise record belongs to an entry that comes af-

ter the cursor location, the current entry is complete and written to disk, and the cursor iterates to the next entry in the Group-By view. Details about the cursor access pattern are given below.

The idea was that by sorting the precise records, all of the entries in each Group-By view in the pipe could be generated in a single scan of the given fact table.

5.2 Comparison between PipeSort and Independent

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 the chains. 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.

5.3 Implementation Details

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 lattice for D. For a given summary table lattice, [4] provides a lower bound on the number of chains in this lattice, which equals the lower bound on the number of summary table groups required for processing. This is also the minimum number of sorts of C. The lower-bound is the length of the longest anti-chain in the summary table lattice (i.e., the "width"). [4] also provides an algorithm to generate this minimal number of chains for a given Group-By view lattice (i.e., summary table lattice). Given a summary table chain, it is straightforward to determine the required sort-order. Thus, we assume we are given as input

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 complete pseudo-code for the Independent algorithm is given in Figure 2. The pseudo-code contains the step "Update cursor on S_i to fact r in S_i that could cover c." This step is implemented in a manner similar to how the current entry in a Group-By relation is determined given the cell-level fact in the PipeSort algorithm in [1].

Theorem 2. *Let $|D|, |C|$ be the number of pages for fact table D and precise 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 requires $7WT|C| + 7T(|D| - |C|)$ 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, we require two passes 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 $+ 4(|D| - |C|)$ I/Os $+ 3W|C|$ I/Os $+ 3(|D| - |C|)$ I/Os ☐

Algorithm 2 Independent Algorithm

```
1:  Method: Independent Allocation
2:  Input: Imprecise database D, Allocation Policy A
3:  Output: Extended Data Model for D which encodes A applied to D
4:  Sort D into Summary Table Order to create summary table lattice
5:  From summary table lattice, find Summary Table Groupings 𝒮, and corresponding Sort-Order Listings L
6:  for (each iteration t) do
7:    for (each summary-table group S ∈ 𝒮) do
8:      Sort C and summary-tables in S into sort-order L
9:      for (each precise fact c in D) do
10:       for (each summary table S_i ∈ S) do
11:         Update cursor on S_i to fact r in S_i that could cover c
12:         if (r ≠ NULL) then
13:           r.delta = r.delta + c.delta
14:   for (each summary table group S ∈ 𝒮) do
15:     for (each precise fact c in D) do
16:       for (each summary table S_i ∈ S) do
17:         Update cursor on S_i to fact r that could cover c
18:         if (r ≠ NULL) then
19:           c.delta = c.delta + r.measure * c.delta / r.delta
```

6. BLOCK ALGORITHM

In this section we describe the Block Algorithm.

6.1 Motivation for Block

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 precise facts in C will be much larger than the total number of imprecise facts in the other summary tables. Sorting 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? 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 (a summary table group corresponds to a summary table lattice path) reduced this to 1 fact for each summary table S_i and the precise summary table C. We referred to this fact as a *cursor*.

Similar to [1], we make the observation that we can process summary tables from different summary table lattice paths using the same sort order if we hold more records in memory for the summary table. Conceptually, this is equivalent to increasing the size of the summary table cursor from a single fact to a contiguous block of records. We refer to such a contiguous block of entries in S as *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. In the section on implementation details below, we will describe how partition sizes are determined.

6.2 Comparison to Overlap Algorithm

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.

Algorithm 3 Block Algorithm

1: *Method: Block Algorithm*
2: *Input:* Imprecise database D, Allocation Policy A
3: *Output:* Extended Data Model for D which encodes A applied to D
4: Sort D into summary table order
5: Given set of summary tables, determine Summary Table Groupings $\mathcal{S}$
6: **for** (each iteration) **do**
7:   **for** (each summary-table group $S \in \mathcal{S}$) **do**
8:     Sort C and summary-tables in S into sort-order L
9:     **for** (each precise fact c in D) **do**
10:       **for** (each summary table $S_i \in S$) **do**
11:         Update cursor on S_i to partition p that could cover c
12:         Find r in p that could cover c
13:         **if** ($r \neq NULL$) **then**
14:           r.delta = r.delta + c.delta
15:   **for** (each summary table group $S \in \mathcal{S}$) **do**
16:     **for** (each precise fact c in D) **do**
17:       **for** (each summary table $S_i \in S$) **do**
18:         Update cursor on S_i to partition p that could cover c
19:         Find r in p that could cover c
20:         **if** ($r \neq NULL$) **then**
21:           c.delta = c.delta + r.measure * $\frac{c.delta}{r.delta}$

6.3 Implementation Details for Block

The complete pseudo-code for Block is given in Figure 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. NEED TO INCLUDE THESE DETAILS. 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.

Since sorting D into summary table order is a step common to all algorithms, we omit counting these I/Os in our analysis. In this analysis, we make the assumption the partition size for each summary table S_i fits into memory.

Theorem 3. *We define $|B|$ as the sum of the partition sizes for all summary tables. Let $|M|$ be the size of the memory buffer. Let T be the number of iterations being performed. Let $W = \lceil \frac{|B|}{|M|} \rceil$. The total number of I/Os performed by the Block algorithm is between $3WT|C| + 3T(|D| - |C|)$ I/Os and $2[3WT|C| + 3T(|D| - |C|)]$.*

THE PROOF OF THE ABOVE IS AS FOLLOWS: Let quantity W is the smallest number of summary table groups such that all summary table partitions fit into memory. The actual number of such summary table groups is an NP-complete problem, 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.

The total number of required I/Os per iteration is given by the following expression.

= $\sum_{i=1}^{W}$[2 scans of C] + 2 scans of each summary table in group i]

7. TRANSITIVE CLOSURE ALGORITHM

In this section we describe the Transitive Closure Algorithm.

7.1 Motivation

The Block Algorithm addresses the locality problem by reducing the number of sorts needed to perform allocation. Since each sort of D is equivalent to several scans, this effectively reduces the number of required scans of D. However, the number of scans *for each iteration* is reduced. For each iteration, Block still performs the same number of I/O operations. In a similar spirit to the spacial locality that Block exploited to reduce the number of scans required for each iteration, we would like to exploit the "iterative" locality, which allows for re-using I/Os across several iterations. Once a fact has been read in memory, we would like to evaluate all iterations of allocation for that fact. Recall that every iteration of allocation generates a new allocation equation. In other words, we would like to evaluate all allocation equations involving the fact *in a single processing pass*.

Consider the following alternative "Divide and Conquer" approach. Assume the given fact table D could be divided into non-overlapping subsets $D_1, D_2, \ldots D_n$, such that allocation weights assigned to facts in subset D_i did not depend on facts in another subset D_j. (We will give details of such a splitting mechanism below). After identifying these subsets of D, each subset is processed independently (i.e., allocation weights are assigned to facts in D_i by performing *all* iterations over D_i required to evaluate the allocation equations).

For the subsets D_i that fit completely into memory, only a single read and write of these records is required *independent of the number of iterations*. We could read each D_j into memory, evaluate all allocation equations, and write D_j out. For subset D_i that do not fit into memory, an efficient algorithm like Block could be executed over the facts in the subset for each iteration. Since the solution for each D_j is independent, once D_j has converged its processing is complete, and it will never have to be considered again for processing other subsets.

7.2 Details of Transitive

First, we define the details of the Overlaps relationship that we use.

Definition 6. (Overlaps Relation and Connected Component) We define the binary relation between two facts *Overlaps(a,b)* as follows: Overlaps(a,b) is true if and only if there exists a fact c such that c is precise and $c \in reg(a) \cap reg(b)$.

For a given fact $c \in D$, we refer to all facts r' in the transitive closure of the Overlaps relationship, $Overlaps^*$, for fact r as a *connected component of* r. Notice that all facts r in a connected component have the same set of facts in $Overlaps^*$. □

Example 5. From the running example, consider facts $p6$ and $p11$. The intersection of the regions for $p6$ and $p11$ (i.e., the cell $\langle MA, Civic \rangle$) contains the precise fact $p1$, thus Overlaps($p6$,$p11$) is true. However, observe that Overlaps($p8$, $p14$) does not hold, since there are no precise facts in $reg(p8) \cap reg(p14)$.

Observe that all facts $p1$ - $p13$ form a single connected component, by the definition given above.

The pseudo-code for the Transitive Algorithm is given in Figure 4.

Algorithm 4 Transitive Algorithm

```
1:  Method: Transitive Algorithm
2:  Input: Imprecise database D, Allocation Policy A
3:  Output: Extended Data Model for D which encodes A applied
    to D
4:  Sort D into summary table order
5:  for (each precise fact c ∈ C) do
6:    initialize overlapSet = NULL
7:    for (each summary table S_i) do
8:      Update cursor on S_i to partition p that could cover c
9:      Find r in p that could cover c
10:     if (r ≠ NULL) then
11:       Add r to overlapSet
12:   Determine set of ccid's assigned to facts in overlapSet).
13:   if (more than 1 ccid assigned to facts in overlapSet) then
14:     Add all ccid's to ccidMapping for smallest ccid in this set
15:     Update the ccid's assigned to each fact in overlapSet to
        this smallest ccid
16:     Write c to relation for smallest ccid (created if not already
        existing)
17: From the (in-memory) ccid-mapping, identify sets of ccids as-
    signed to the same connected component. Assign each such set
    a unique ccidSetId
18: for (each ccidSetId ccidSid) do
19:   Find the sum of connected component relation sizes for ccid
      set ccidSid.
20:   Call this quantity ccidSidSize
21:   (This is the true size of the connected component)
22:   if (ccidSidSize < M) then
23:     scan all relations corresponding to ccid's in ccidSid into
        memory
24:     Iterate over these facts until converged
25:     Write all facts out to new relation with identifier ccidSid
26:   if (ccidSetIdSize > M) then
27:     create new relation ccidSid by concatenating together all
        relations corresponding to ccid's in ccidSid
28:     sort relation ccidSid into summary table order.
29:     for (each iteration t) do
30:       Perform Block over the relation ccidSid
```

7.3 Cost Analysis for Transitive

Since the initial sort of D into summary table order is common to all algorithms, we omit its cost from the analysis.

The two main parts of the Transitive Closure Algorithm are the *component identification step*, during which connected components are identified, and the *component processing step*, during which connected components identified in the first step are processed. From the algorithmic description, it should be clear I/O costs for these steps can be analyzed independently.

First, we analyze the component identification step. Consistent with the notation used in the analysis for Block, we define $|B|$ to be the sum of the current partition size for each summary table S_i, and $|M|$ to be the size of the memory buffer. There are two cases to consider. In both cases, each fact (both imprecise and precise) is written out to the appropriate connected component relation on disk.

1. $|B| < |M|$, *the current partition for each S_i can simultaneously be held in memory.* During the scan of C, we only process each partition of S_i once. The argument is identical to the one made for processing during the Block algorithm. We perform a read and write I/O for each fact in D, for a total of $2|D|$ I/Os.

2. $|B| > |M|$, *the current partition for each S_i can not be held together in memory.* For each precise record c, we require access to the current partition for each S_i in memory.

   During the scan of C, for each precise record c, all current summary table partitions must examine blocks for each precise fact. Let $|c|$ be the number of precise records in C. The number of required I/Os is $|c||B| + |C|$.

For the second case (i.e., when $|B| > |M|$), the required number of I/Os is proportional to the number of precise facts $|c|$, which is clearly an infeasible. For the I/O analysis of processing the connected components, we assume we are in case 1. Since a fact can belong to at most one connected component, the size of all connected components is still $|D|$ pages. (A reasonable implementation would buffer output and place several small connected components on the same disk page, thus it is reasonable to assume disk pages remain packed). We refer to a connected component that fit completely into memory (i.e., has less that $|M|$ pages of facts) as a *small connected component*. A connected component with more than $|M|$ pages of records is called a *large connected component*.

Assume we are performing k iterations. Let the facts belonging to "large" connected components (i.e., connected components whose size is greater than $|M|$) occupy $|L|$ pages of D. For each iteration, the large component is processed using the Block algorithm, requiring 3 I/Os per page per iteration, for a total of $3k|L|$ I/Os for all iterations over all large components. Since this analysis assumes we are in case 1 of the splitting algorithm, we assume all current partitions for each S_i in D fit into memory. Since each large connected component is a subset of D, this property holds for all large connected components.

Before the initial iteration, each large component must be sorted into summary table order. The output from the component generation step has no means to guarantee this property holds. This requires a total of $4|L|$ I/Os. The total cost for processing all small connected components is $2(|D| - |L|)$, since facts in small components only need to be 2 I/Os each(read and written exactly once).

From the above analysis, we conclude the I/O performance of the component processing step for Transitive Closure dominates Block. The degree of improvement is determined by the number

of facts in small connected components. By algebraic manipulation of the above formulas, the improvement can be quantified by the expression $[(3k-1)(|D|-|L|)]$, where k is the number of iterations. In the worst case, every record is part of some large connected component (i.e., $|D|=|L|$), and there is no improvement.

The cost for the entire Transitive Closure algorithm (both steps) is lower than Block when the savings from component processing step is greater than the component identification step. For the case when the splitting step is feasible (Case 1), the expression is $2|D| < [(3k-1)(|D|-|L|)]$.

8. EXPERIMENTS

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

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

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