Finding Intensional Knowledge of Distance-Based Outliers

Edwin M. Knorr and Raymond T. Ng
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4 Canada

Abstract

Existing studies on outliers focus only on the *identification* aspect; none provides any *intensional knowledge* of the outliers—by which we mean a *description* or an *explanation* of why an identified outlier is exceptional. For many applications, a description or explanation is at least as vital to the user as the identification aspect. Specifically, intensional knowledge helps the user to: (i) evaluate the validity of the identified outliers, and (ii) improve one's understanding of the data.

The two main issues addressed in this paper are: *what kinds* of intensional knowledge to provide, and *how to optimize* the computation of such knowledge. With respect to the first issue, we propose finding *strongest* and *weak* outliers and their corresponding structural intensional knowledge. With respect to the second issue, we first present a naive and a semi-naive algorithm. Then, by means of what we call *path* and *semi-lattice* sharing of I/O processing, we develop two optimized approaches. We provide analytic results on their I/O performance, and present experimental results showing significant reductions in I/O, and significant speedups in overall runtime.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Knowledge discovery tasks can be classified into four general categories: (a) dependency detection, (b) class identification, (c) class description, and (d) exception/outlier detection. The first three categories of tasks correspond to patterns that apply to many objects, or to a large percentage of objects, in the dataset. Most research in data mining (e.g., association rules and variants [AIS93, BMS97, NLHP98, SBMU98], data clustering [KR90], and classification [AGI+92, BFOS84]) belongs to these three categories. In contrast, the fourth category focuses on only a very small percentage of data objects. While such objects are sometimes ignored or treated as "noise", we note that "one person's noise is another person's signal." In other words, for numerous knowledge discovery applications, the rare events can be more interesting than the common events. Sample applications include the detection of credit card fraud and the monitoring of criminal or suspicious activities (e.g., [TN98]).

1.1 Related Work

Existing approaches to outlier detection can be broadly classified into three categories. The first category is *distribution-based*, which relies on fitting the data with standard statistical models. (See [BL94] for a comprehensive treatment.) However, distribution-based approaches are mainly univariate in nature, and require extensive and expensive testing to find a distribution to fit the data, thus making such approaches unsuitable for most data mining applications.

The second category is *depth-based*, which relies on organizing the data objects in some k-D space. Based on some definition of depth (e.g., [Tuk77]), data objects are organized in layers in the data space, with the expectation that shallow layers are more likely to contain outlying data objects than are the deep layers. Depth-based approaches avoid the aforementioned problem of distribution fitting, and conceptually allow multi-dimensional data objects to be processed. Although there are efficient techniques for 2-D spaces

Player Name	Power-play Goals	Short-handed Goals	Game-winning Goals	Game-tying Goals	Games Played
MARIO LEMIEUX	31	8	8	0	70
JAROMIR JAGR	20	1	12	1	82
JOHN LECLAIR	19	0	10	2	82
ROD BRIND'AMOUR	4	4	5	4	82

Figure 1: NHL Players' Statistics: Outliers Identified

[JKN98, RR96], depth-based approaches do not scale up well with the dimensionality k. Specifically, depth-based approaches are lower bounded in complexity by k-D convex hull computation, i.e., $\Omega(n^{k/2})$, where n is the number of data objects.

The third category is *distance-based.* As we introduced in [KN98],

An object O in a dataset is a $DB(p, D)$-outlier if at least a fraction p of the other objects in the dataset lies greater than distance D from O.

We showed that the notion of $DB(p, D)$-outliers generalizes many of the distribution-based outliers discussed above, and we developed (i) Algorithm NL with complexity $O(k\ n^2)$—thus, making it far more attractive than the depth-based approaches for k-D datasets with $k \geq 4$; and (ii) Algorithm CELL [1] with a complexity linear on n, and a guarantee of no more than 3 passes over the data. However, the latter Algorithm CELL is exponential on k, and is recommended only for smaller values of k (i.e., $k \leq 4$). In [TN98], we documented a case study that successfully applies the distance-based outlier methodology to a video surveillance situation, in which the dimensionality of the dataset exceeds 20.

1.2 Contributions of This Paper

All the studies on outliers research focus only on *identification*; none is able to provide any *intensional knowledge* of the outliers—by which we mean a *description* or an *explanation* of why an identified outlier is exceptional. For many applications, the description/explanation aspect is at least as vital to the user as the identification aspect. Intensional knowledge helps the user to: (i) evaluate the validity or the credibility of the identified outliers, and (ii) more importantly, improve the user's understanding of the data.

Consider an example based on 1995-96 National Hockey League players' statistics. (Such statistics can be found at http://nhlstatistics.hypermart.net, among many other sites.) For certain parameter values of p and D and a specific choice of a distance function, the four players in Figure 1 are identified as exceptional (among 855 players) in the 5-D space of power-play goals, short-handed goals, game-winning goals, game-tying goals, and games played. Note that the numbers in the table cannot directly pinpoint the strengths

[1] In [KN98], Algorithm CELL was called FindAllOutsM or FindAllOutsD for memory or disk-resident versions, respectively. For convenience, we use the name CELL. We deal exclusively with disk-resident data in this paper.

```
MARIO LEMIEUX:
  (i)  An outlier in the 1-D space of Power-play goals
  (ii) An outlier in the 2-D space of Short-handed goals and
       Game-winning goals
       (No player is exceptional on Short-handed goals alone;
        No player is exceptional on Game-winning goals alone.)
ROD BRIND'AMOUR:
  (i)  An outlier in the 1-D space of Game-tying goals
JAROMIR JAGR:
  (i)  An outlier in the 2-D space of Short-handed goals and
       Game-winning goals
       (No player is exceptional on Short-handed goals alone;
        No player is exceptional on Game-winning goals alone.)
  (ii) An outlier in the 2-D space of Power-play goals and
       Game-winning goals
       (But for Power-play goals alone, the current player is
        dominated by another and is not exceptional.)
JOHN LECLAIR:
  (i) An outlier in the 2-D space of Game-winning goals and
      Game-tying goals
      (But for Game-tying goals alone, the current player is
       dominated by another and is not exceptional.)
```

Figure 2: NHL Players' Statistics: Intensional Knowledge Provided

or greatness of these players with respect to all other NHL players. Specifically, the intensional knowledge described in Figure 2 cannot be derived directly from the numbers. While we shall discuss in detail in Section 2 the nature of the provided intensional knowledge, it suffices to say that the additional explanations give new and valuable insights about the dataset in general and about the outliers in particular.

The two main issues addressed in this paper are: *what kinds* of intensional knowledge to provide, and *how to optimize* the computation of such knowledge. Specifically:

- We define in Section 2 two notions of outliers and the corresponding intensional knowledge. The two notions are *strongest* outliers and *weak* outliers. To illustrate, Jaromir Jagr in Figure 2 is: (i) a strongest outlier in the 2-D combination of short-handed goals and game-winning goals, but (ii) only a weak outlier in the 2-D combination of power-play goals and game-winning goals.

- We develop in Section 3, a naive and a semi-naive algorithm for computing strongest and weak outliers and the corresponding intensional knowledge. While the naive algorithm conducts its processing in a bottom-up fashion over an appropriate lattice, the semi-naive algorithm starts at a certain intermediate level of the lattice, and "drills down" to lower levels only if necessary.

- We show in Section 4 that, instead of processing one node of the lattice at a time, it is possible to

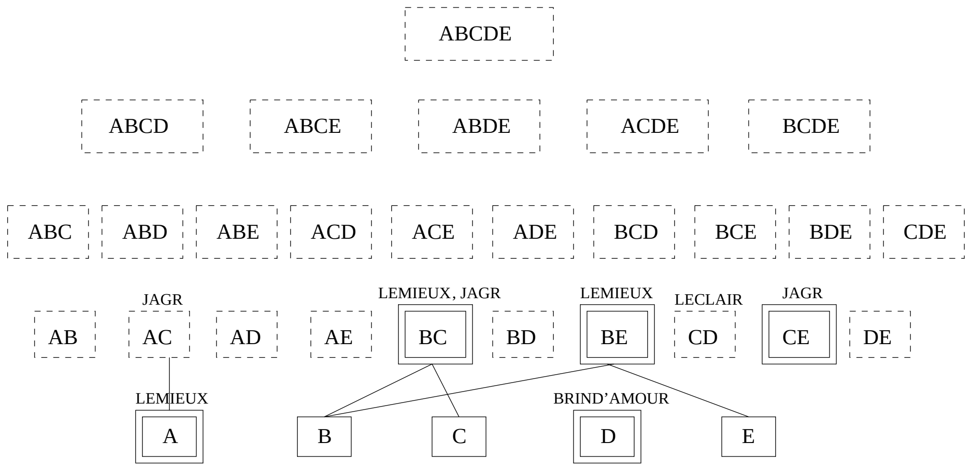


Figure 3: Intensional Knowledge in a Lattice Representation

process nodes in groups, thereby sharing the I/O's among the nodes. We show that effective sharing of I/O's can be achieved in two flavours: a *path* mode and a *semi-lattice* mode. We present analytic results guaranteeing their I/O performance. Finally, we give experimental results showing how much improvement these optimizations can yield.

2 Strongest and Weak Outliers and the Corresponding Intensional Knowledge

In this section, we first introduce the notions of strongest and weak outliers, and show examples. Then, we offer a few justifications of why we opt to work with these notions.

2.1 Definitions

Given an object P which has been identified to be exceptional in some attribute space $\mathcal{A}_P$ (containing possibly many attributes/dimensions), two items of knowledge that would be of interest to the user are:

I. What is the smallest set of attributes to explain why P is exceptional?

II. Is P "dominated" by other outliers?

With respect to question (I), given $\mathcal{A}_P$, we seek to find all the minimal subsets of attributes in which P is exceptional. We have the following definition.

Definition 1 Suppose that P is an outlier in the attribute space $\mathcal{A}_P$. P is a *non-trivial* outlier in $\mathcal{A}_P$ if P is not an outlier in any subspace $\mathcal{B} \subset \mathcal{A}_P$. □

Note that a subspace's points are simply a projection of points of a given/original space onto a subset of its attributes. While question (I) focuses on the examination of P alone, question (II) concerns the evaluation of P against other outliers. With respect to question (II), we seek to find the minimal (sub)sets of attributes in which there is an outlier—*any* outlier, be it P, Q, etc. We have the following definition.

Definition 2 Let $\mathcal{A}_P$ be an attribute space containing one or more outliers. $\mathcal{A}_P$ is called a *strongest outlying space* if no outlier exists in any subspace $\mathcal{B} \subset \mathcal{A}_P$. Furthermore, any P that is an outlier in $\mathcal{A}_P$ is called a *strongest* outlier. □

One reason for why a strongest outlying space $\mathcal{A}_P$ is noteworthy is that this is the first time that $\mathcal{A}_P$ and any of its subspaces contain an outlier. It follows immediately from the above definitions that if P is a strongest outlier, then P must be non-trivial. To completely classify non-trivial outliers, we have the following definition.

Definition 3 Suppose that P is a non-trivial outlier in the attribute space $\mathcal{A}_P$. Then if P is not a strongest outlier, P is called a *weak* outlier. □

In summary, we differentiate among three kinds of outliers: *strongest*, *weak*, and *trivial* outliers. Trivial outliers are the least interesting because they convey no new information than what was already observed or reported for some subspace of the given space.

2.2 NHL Players Example Revisited

According to the above definitions, we can represent the intensional knowledge given in Figure 2 using the lattice shown in Figure 3. This is the attribute lattice formed by the 5 attributes in our example, with the attributes denoted as A, B, C, etc. We use single-lined rectangles to denote attribute spaces that do not contain any outlier; we use double-lined rectangles to denote strongest outlying spaces; and we use dashed rectangles to denote spaces that cannot contain strongest outliers but *may* contain weak outliers. All strongest outliers are shown, but for simplicity, not all weak outliers and not all edges between nodes are shown. We have labelled the nodes of the lattice with the outliers that first appear at a given node (and nowhere below it). Corresponding to Figure 2, Mario Lemieux is a strongest outlier in: the 1-D space of power-play goals (denoted as "A" in the figure), and the 2-D space of short-handed goals and game-winning goals (denoted as "BC"). In the latter space, Jaromir Jagr is also a strongest outlier. However, Jagr is only a weak outlier in the 2-D space "AC" of power-play goals and game-winning goals, because Jagr is being dominated by Lemieux in "A". This example shows that a data object can be a strongest and/or weak outlier in multiple spaces at the same time.

Although this is only a very small example, it is easy to see that such a categorization of outliers conveys far more information to a user than does a single, unqualified list of all outliers for the largest space only. Thus, a user who is not hockey literate can actually see which players dominate which attributes. Even domain experts may be surprised to find players who surface as outliers in unusual combinations of attributes. As our earlier work [KN98] pointed out, outliers need not be those observations which have extreme values. For example, it is quite possible that a player with "average" values is an outlier, simply because that player is so different from the rest of the players in the dataset.

In a very large dataset containing many attributes, there may be many outliers, but some of those outliers may be strongest outliers which noticeably stand out among the larger pool of outliers. Such outliers may get closer scrutiny, as might be the case involving "worst offenders" in a fraud detection application.

2.3 Justification of Our Choices of Intensional Knowledge

Earlier, we pointed out that it is valuable to provide explanations/descriptions about the identified outliers, but explanations can clearly come in many different forms. Below, we discuss why we opt to work with the intensional knowledge associated with strongest and weak outliers.

- First, we hope the NHL example has served to argue that the knowledge corresponding to strongest and weak outliers is intuitive and meaningful to the user, and is otherwise not immediately derivable by just using the values of the attributes in the space $\mathcal{A}_P$.
- Second, the notions of strongest and weak outliers are close in spirit with the subspace clustering problem addressed by Agrawal *et al.* [AGGR98]. They obtain not only the clusters in the original attribute space, but also the clusters in the subspaces. Our work is different from their work in at least two ways. First, we go beyond minimal description length (i.e., non-trivial outliers) and seek to find out whether a non-trivial outlier is dominated by others—in the sense formalized by strongest outliers. Imposing a partial order denoting the dominance or strength of non-trivial outliers is well-suited to our task. Second, finding strongest and weak outliers is computationally different from subspace clustering. The remaining sections of this paper will illustrate why.
- Finally, there are numerous forms of explanations possible. In general, there is the tradeoff between what the explanation says and how long it takes to compute it. For the two reasons given above, we believe that the explanation offered by strongest and weak outliers is meaningful. We will show below that their computation is highly optimizable.

In closing this section, we comment that we can extend the definitions given in Section 2.1 by defining the notion of *top-u* strongest or weak outliers, where we rank strongest or weak outliers P in ascending order of cardinalities of their associated space $\mathcal{A}_P$. That is, we prefer an outlier that can be explained with fewer attributes. The top-u notion, in practice, is particularly natural for weak outliers. This is because, given their nature, weak outliers can be more numerous than strongest outliers. The top-u notion helps to avoid a situation whereby too many outliers are returned than the user cares for, and yet the user is charged for the entire computation. In the remainder of this paper, though, we give algorithms that compute *all* strongest and weak outliers; it should be easy to see how they can be modified to give a top-u list.

3 Two Simpler Algorithms for Finding Strongest and Weak Outliers

In this section, we present a naive algorithm, called UpLattice, for finding all strongest and/or weak outliers. Then, we present a more intelligent algorithm, called JumpLattice. Before doing so, however, we provide a short summary in Section 3.1 of the algorithms developed and evaluated in [KN98] for finding outliers in a particular space. Readers familiar with that paper can skip the summary.

3.1 Background Summary: Algorithms for Finding Outliers in a Particular Space

Given a particular space $\mathcal{A}$, with cardinality (dimensionality) k, we considered in [KN98] primarily two algorithms [2] for finding all outliers in that space. First, there is Algorithm NL, which for a given amount of buffer space, uses a nested loop algorithm to find all outliers. In the worst case, NL checks each pairwise distance between two objects/tuples, and thus has a complexity of $O(k\, n^2)$. See [KN98] for details of the algorithm. Experimental results show that NL is the faster algorithm for $k \geq 5$.

[2] Actually, we evaluated another strategy, which we call Algorithm INDEX, that is based on a multi-dimensional index such as an R*-tree. Because this algorithm is dominated by CELL for small values of k, and by NL for the remaining values of k, we skip that algorithm here.

Second, there is Algorithm CELL, which is based on building an optimized cell structure. The general idea is as follows. Let D be the radius of the local neighbourhood (i.e., the D value selected for $DB(p, D)$-outliers). The k-D space is divided into cells with length equal to $\frac{D}{2\sqrt{k}}$ along each dimension. Each tuple is then quantized (mapped) to an appropriate cell. Because of the carefully chosen size of each cell, we can first eliminate cells, called "red" cells, that contain too many tuples quantized to them, and also those cells which are immediate neighbours of red cells. Thus, very quickly, a significant number of cells—and the large number of tuples they contain—are pruned. What remain are called "white" cells, containing "white" tuples. These are tuples P that require explicit distance calculations with each tuple Q in certain neighbouring cells. This phase, as expected, can require many I/O operations, and makes the algorithm essentially I/O-dominant. While readers are referred to [KN98] for more details, because the focus of this paper is on I/O optimizations, we summarize below the four key phases of I/O operations:

- Phase 1: Make one pass over the entire dataset to quantize the tuples into cells, from which the "non-white" cells are eliminated.
- Phase 2: Read the pages that contain some white tuples P. We call such pages *Class I* pages.
- Phase 3: Read the pages that do not contain any white tuples but contain non-white tuples Q needed by some white tuple P for tuple-by-tuple processing. We call such pages *Class II* pages.
- Phase 4: Repeat Phase 2.

In [KN98], we showed why the above four phases of I/O are all the I/O operations that are needed. Moreover, we have the following performance guarantee.

Lemma 1 ([KN98]) (1) Class I pages and Class II pages are mutually exclusive. (2) The total number of pages read is equal to $M+2M_1+M_2$, where M, M_1 and M_2 are the total number of pages of the entire dataset, of Class I pages, and of Class II pages, respectively. (3) Each page in the dataset is guaranteed to be read at most 3 times. □

In practice, most pages are read only once or twice. The complexity of CELL is $O(c^k\, n)$, and experimental results show that Algorithm CELL is the faster algorithm for $k \leq 4$.

3.2 A Naive Algorithm: Procedure UpLattice

As suggested in Figure 3, the computation of strongest and weak outliers can be conducted over the lattice formed by the attributes. As is done in association rule mining [AIS93], subspace clustering [AGGR98], and functional dependencies detection [HKPT98], a standard algorithmic strategy is to compute or traverse the lattice in a bottom-up, level-wise fashion. That is to say, we first find outliers in the 1-D spaces, then in the 2-D spaces, and so on. This simple algorithm is outlined in Figure 4.

Procedure UpLattice($\mathcal{A}$)

1 Insert into queue Q all the subsets of attribute set $\mathcal{A}$ in ascending order of cardinalities.

2 While Q is not empty {

  2.1 Remove the subset of attributes at the front of Q. Let that be $\mathcal{A}'$.

  2.2 Call Procedure FindOutliersInNode($\mathcal{A}'$).

  2.3 If there is some outlier found above {

    2.3.1 If only strongest outliers are to be found, remove from Q all supersets of $\mathcal{A}'$. } }

Procedure FindOutliersInNode($\mathcal{A}'$)

1 If $|\mathcal{A}'| \leq 4$, call Procedure CELL($\mathcal{A}'$)

2 Else call Procedure NL($\mathcal{A}'$).

Procedure JumpLattice($\mathcal{A}$, k)

1 Insert into Q all the subsets of $\mathcal{A}$ (which have cardinalities $\geq k$) in ascending order of cardinalities.

2 While Q is not empty {

  2.1 Remove the subset of attributes at the front of Q. Let that be $\mathcal{A}'$.

  2.2 Call Procedure FindOutliersInNode($\mathcal{A}'$).

  2.3 If there is some outlier found above {

    2.3.1 If only strongest outliers are to be found, remove from Q all supersets of $\mathcal{A}'$.

    2.3.2 If $|\mathcal{A}'| = k$, call Procedure DrillDown($\mathcal{A}'$).} }

Procedure DrillDown($\mathcal{A}'$)

1 Insert into queue Q the proper subsets of $\mathcal{A}'$ in *descending* order of cardinalities.

2 While Q is not empty {

  2.1 Remove the subset of attributes at the front of Q. Let that be $\mathcal{B}$.

  2.2 Call Procedure FindOutliersInNode($\mathcal{B}$).

  2.3 If $|\mathcal{B}| > 1$ and if no outlier is found in $\mathcal{B}$, remove from Q all *subsets* of $\mathcal{B}$. } }

Figure 4: Skeletons for UpLattice and JumpLattice

There are three aspects worth noting about Procedure UpLattice as shown in the figure. First, it should be clear from Definition 2 that once the space $\mathcal{A}'$ is found to contain some outlier, any superset of $\mathcal{A}'$ cannot contain any strongest outlier. This is the purpose of Step 2.3, and this is one of the few differences between the computation of strongest and weak outliers. Note that if only strongest outliers are sought, and in the very rare event that all strongest outliers appear at level 1, then UpLattice is very efficient. Second, if the skeleton shown in Figure 4 is to find weak outliers, strictly speaking, there should be the additional step to screen out trivial outliers after Step 2. We do not include the step here because all the algorithms considered in this paper differ mainly in how the lattice is traversed or grouped. Thus, Figure 4 only concentrates on this aspect. Third, as summarized in Section 3.1, the sub-procedure FindOutliersInNode($\mathcal{A}'$) chooses between Algorithms CELL and NL, based on the cardinality of $\mathcal{A}'$.

3.3 Procedure JumpLattice

A key problem with Procedure UpLattice is that there could be few outliers (if any) found in the smallest dimensional spaces. In other words, a bottom-up strategy may take a lot of wasted effort before a space containing outliers is encountered. It would be nice to be able to jump to some node in the intermediate levels of the lattice, instead of climbing slowly from the bottom. The advantage is that if there is no outlier in some intermediate level node $\mathcal{A}'$, then none of the remaining $2^k - 2$ non-empty subsets of $\mathcal{A}'$ (where $k = |\mathcal{A}'|$) needs to be considered anymore. Figure 4 shows a skeleton of Procedure JumpLattice. Note that the only differences between JumpLattice and UpLattice are: (i) Step 2.3.2, which deals with the situation when a space $\mathcal{A}'$ of cardinality k contains some outlier, and (ii) the level k at which to begin processing. (Note that JumpLattice is a strict generalization of UpLattice because for $k = 1$, the former is reduced to the latter.) We examine these differences below.

3.3.1 Procedure DrillDown

When the space $\mathcal{A}'$ of cardinality k contains some outlier, what needs to be done, for both strongest and weak outliers, is to consider the proper subsets of $\mathcal{A}'$. This is outlined in Procedure DrillDown in Figure 4. Specifically, subsets $\mathcal{B}$ of $\mathcal{A}'$ are processed one-by-one by calling the same Procedure FindOutliersInNode($\mathcal{B}$). If no outlier is found in $\mathcal{B}$, then the "drill down" can omit the subspaces of $\mathcal{B}$. The fact that no outlier exists in $\mathcal{B}$ suffices to guarantee that no outlier exists in any of its subspaces.

3.3.2 Choice of k for JumpLattice

In the above discussion, we motivated why JumpLattice can be useful. However, if JumpLattice starts with $\mathcal{A}'$ at a level k that turns out to be too high, and $\mathcal{A}'$ *does* contain some outlier, then the subsets of $\mathcal{A}'$ need to be examined. In the case that the task is to find strongest outliers or the top-u strongest/weak outliers, processing $\mathcal{A}'$ can be completely wasteful (i.e., both time consuming and unnecessary) in the event that there is a subset containing an outlier. So the main question is: What should the value of k be? Unfortunately, we do not know of any reliable analysis that can be done to predict an optimal value for every situation. We do, however, offer the following heuristic.

Figure 5 shows the relative amount of (CPU+I/O) time taken to find all outliers in a particular space $\mathcal{A}'$ with all parameters set to identical values, except for the dimensionality $k = |\mathcal{A}'|$ (in effect, running Procedure FindOutliersInNode for varying k). This graph is based on using the best algorithms among those summarized in Section 3.1. The results were experimentally obtained and are representative of numerous other cases. It is clear from the graph that as k increases from 1 to 3, the "marginal cost" is relatively small. However, as k increases to 4 and beyond, the "marginal cost" becomes substantial for each increment in k. Thus, our heuristic is:

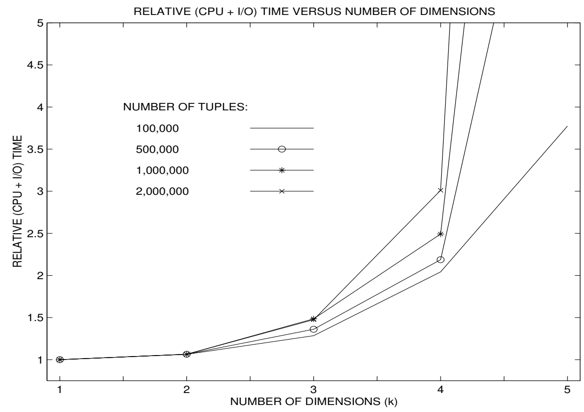


Figure 5: Heuristic Selection of k for JumpLattice

In the absence of any information predicting the value of k, [3] we pick k to be 3.

In Section 5, we present experimental results comparing JumpLattice with UpLattice. We will report exclusively the results of JumpLattice with k set to 3. Due to the lack of space, results for other values of k are not shown; however, for the conclusions we want to draw, the results with $k = 3$ are sufficiently representative.

4 Grouped Processing of Multiple Nodes and Performance Analysis

So far, we have considered two ways of processing the nodes in the lattice to find strongest and/or weak outliers, but in both cases, nodes are processed or examined one at a time (i.e., via Procedure FindOutliersInNode). The objective of this section is to show how multiple nodes, suitably selected, can be grouped and processed together. We provide proof of correctness and performance guarantees.

4.1 Grouping Two Nodes Satisfying an Edge Relationship

We begin by considering the simultaneous processing of two nodes/spaces using Algorithm CELL, and later

[3] We only recommend $k = 3$ in the event that no other information is available, and there are indeed situations whereby other values of k could be more beneficial. One example is a situation in which the user runs the algorithms with varying parameter values. Knowing which subspaces contain outliers, given the current set of parameter values, the value of k can be set up more intelligently by comparing the new set of parameter values with the previous set. For example, if the new set of parameter values has the effect of decreasing the number of outliers, then k should be set higher. We do not pursue the issue of incremental processing here, but simply comment on the utility of the general JumpLattice Procedure.

we will generalize this notion to many nodes.

Definition 4 Given two nodes $\mathcal{A}$, $\mathcal{B}$, we say that $\mathcal{A}, \mathcal{B}$ satisfy a (directed) *edge* relationship, denoted as $\langle \mathcal{A}, \mathcal{B} \rangle$, if $\mathcal{A} \supset \mathcal{B}$ and $|\mathcal{A}| = |\mathcal{B}| + 1$. That is, an edge exists between the two nodes in the lattice. □

For the example shown in Figure 3, $\mathcal{A} = \{A, B, C\}$ and $\mathcal{B} = \{A, B\}$ satisfy an edge relationship. Recall from Section 3.1 that Algorithm CELL requires four I/O phases. The task here is to examine how each of the four phases can be extended to process the two nodes simultaneously. During Phase 1 (quantizing the tuples into cells and eliminating "non-white" cells), processing two nodes together is easy, because separate cell structures can be maintained. The situation is, however, very different for the other phases. We carefully consider those phases, below.

4.1.1 Combined Reading of Class I Pages (Phase 2)

By the definition given in Section 3.1, Class I pages are data pages that contain some white tuples. For a given white tuple P, we use the notation $PgI(P)$ to denote the single data page that contains P. For a space $\mathcal{A}$, we use the notation $WT_{\mathcal{A}}$ to denote the set of all white tuples in $\mathcal{A}$. Then, to generalize, we use the notation $PgI(S)$ to denote the set of data pages that contain tuples in the set S, that is, $PgI(S) = \cup_{P \in S}\ PgI(P)$. The following result identifies a key relationship between white tuples for two nodes satisfying an edge relationship.

Lemma 2 Given the edge relationship $\langle \mathcal{A}, \mathcal{B} \rangle$, it is the case that: P is a white tuple in $\mathcal{B} \Rightarrow P$ is a white tuple in $\mathcal{A}$. □

While details of a proof are left to [KN99], the basic rationale behind this lemma is that the distance between P and any tuple Q grows from $\mathcal{B}$ to $\mathcal{A}$. If there are not enough tuples surrounding P in a fixed-radius local neighbourhood in $\mathcal{B}$, there cannot be enough tuples surrounding P in the same-radius neighbourhood in $\mathcal{A}$. It is easy to see that the converse of the lemma is not true. With the above lemma, we have the following result.

Lemma 3 Given the edge relationship $\langle \mathcal{A}, \mathcal{B} \rangle$, it is the case that: the set of data pages containing white tuples in $\mathcal{B}$ is contained in the corresponding set in $\mathcal{A}$, that is, $PgI(WT_{\mathcal{B}}) \subseteq PgI(WT_{\mathcal{A}})$. □

The rationale behind the above lemma is that from Lemma 2, it is necessary that the set of white tuples in $\mathcal{B}$ is contained in the corresponding set in $\mathcal{A}$, i.e., $WT_{\mathcal{B}} \subseteq WT_{\mathcal{A}}$. The significance of the above lemma is that as far as Phase 2 of Algorithm CELL is concerned, it suffices to read in Class I pages for $\mathcal{A}$ for the processing of *both* $\mathcal{A}$ and $\mathcal{B}$, as summarized below.

Corollary 1 Given $\langle \mathcal{A}, \mathcal{B} \rangle$, the combined set of Class I pages to process both spaces simultaneously is given by: $PgI(\ WT_{\mathcal{A}} \cup WT_{\mathcal{B}}\ ) = PgI(WT_{\mathcal{A}})$. □

4.1.2 Combined Reading of Class II Pages (Phase 3)

Next, we consider Class II pages. We use the notation $PgII(P, \mathcal{A})$ to denote the set of data pages containing non-white tuples needed to check whether white tuple P is an outlier in $\mathcal{A}$. [4] Then, to generalize, we use the notation $PgII(S, \mathcal{A})$ to denote $\cup_{P \in S}\ PgII(P, \mathcal{A})$.

Lemma 4 Given the edge relationship $\langle \mathcal{A}, \mathcal{B} \rangle$, it is the case that: $PgII(P, \mathcal{A}) \subseteq PgII(P, \mathcal{B})$, for white tuples P in $\mathcal{B}$ (and therefore in $\mathcal{A}$). □

The rationale behind the lemma is that for checking whether P is an outlier, because the distance between P and any other tuple Q grows from $\mathcal{B}$ to $\mathcal{A}$, there are possibly more tuples Q to be examined in $\mathcal{B}$ than in $\mathcal{A}$ within a fixed-radius neighbourhood. When compared with Lemma 3 for Class I pages, the above lemma points out that—for white tuples P in both $\mathcal{B}$ and $\mathcal{A}$—the containment relationship for Class II pages is *opposite* to that for Class I pages. And as a counterpoint to Corollary 1, we have the following corollary. It says that for Phase 3 of Algorithm CELL, it suffices to read in the Class II pages for $\mathcal{B}$, as well as the Class II pages for all the white tuples in $\mathcal{A}$ but not in $\mathcal{B}$.

Corollary 2 Given $\langle \mathcal{A}, \mathcal{B} \rangle$, the combined set of Class II pages to process both spaces simultaneously is given by: $PgII(WT_{\mathcal{A}}, \mathcal{A}) \cup PgII(WT_{\mathcal{B}}, \mathcal{B}) = PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A}) \cup PgII(WT_{\mathcal{B}}, \mathcal{A}) \cup PgII(WT_{\mathcal{B}}, \mathcal{B}) = PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A}) \cup PgII(WT_{\mathcal{B}}, \mathcal{B})$. □

4.1.3 I/O Performance Guarantee: Number of I/O's Saved

By now, it should be clear that the same four phases of Algorithm CELL, as shown in Section 3.1, are sufficient to simultaneously process two nodes satisfying an edge relationship—so long as the Class I and II pages are modified as given in the previous lemmas. Below, we consider two kinds of I/O performance guarantees for the shared processing. The first guarantee applies to the number of I/O's that can be saved.

Lemma 5 Compared with processing $\mathcal{A}$ and $\mathcal{B}$ separately, grouped processing of $\langle \mathcal{A}, \mathcal{B} \rangle$ saves at least $M + 2M_{1,\mathcal{B}}$ page reads.

Proof Sketch: Recall from Lemma 1 that if we are to process $\mathcal{A}$ and $\mathcal{B}$ separately, the total number of I/O's will be $(M + 2M_{1,\mathcal{A}} + M_{2,\mathcal{A}})+\ (M + 2M_{1,\mathcal{B}} + M_{2,\mathcal{B}})$,

[4] For Class I pages, it is not necessary to use the space as a parameter, because regardless of which space we are considering, it is the same data page that contains the tuple P. For Class II pages, however, the space $\mathcal{A}$ may make a difference.

where $M_{1,\mathcal{S}}$ and $M_{2,\mathcal{S}}$ denote the number of Class I and II pages for space $\mathcal{S}$, respectively. Based on the two corollaries above, the total number of I/O's required by the grouped processing of $\langle \mathcal{A}, \mathcal{B} \rangle$ is upper bounded by $(M + 2M_{1,\mathcal{A}} + M_{2,\mathcal{B}} + |PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A})|)$. The last term is upper bounded by $|PgII(WT_{\mathcal{A}}, \mathcal{A})| = M_{2,\mathcal{A}}$. Thus, the total number of I/O's saved is at least $M + 2M_{1,\mathcal{B}}$. □

4.1.4 I/O Performance Guarantee: Reads per Page

The second I/O performance guarantee applies to the number of reads per page.

Lemma 6 When applying CELL to process two spaces $\langle \mathcal{A}, \mathcal{B} \rangle$ simultaneously, each page is guaranteed to be read at most 4 times. □

Recall from Lemma 1 that when CELL is applied to one space only, Class I and Class II pages are mutually exclusive. This is the reason why even though there are four different phases involving I/O, each page is guaranteed to be read at most 3 times. The situation for two spaces satisfying an edge relationship is different, because the combined Class I pages as given in Corollary 1 and the combined Class II pages as given in Corollary 2 are not guaranteed to be mutually exclusive. To see this, let us consider the intersection between $PgI(WT_{\mathcal{A}})$ and $PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A}) \cup PgII(WT_{\mathcal{B}}, \mathcal{B})$ by considering two cases: (a) $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A})$, and (b) $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{B}}, \mathcal{B})$.

(a) Since (i) $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{A}}, \mathcal{A}) = \emptyset$, and (ii) $PgII(WT_{\mathcal{A}}, \mathcal{A}) \supseteq PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A})$, it is necessary that $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{A}} - WT_{\mathcal{B}}, \mathcal{A}) = \emptyset$. There is mutual exclusion here.

(b) However, for the intersection $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{B}}, \mathcal{B})$, we know that (i) $PgI(WT_{\mathcal{B}}) \cap PgII(WT_{\mathcal{B}}, \mathcal{B}) = \emptyset$, but (ii) $PgI(WT_{\mathcal{A}}) \supseteq PgI(WT_{\mathcal{B}})$. Therefore, it is *not* sufficient to conclude that $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{B}}, \mathcal{B}) = \emptyset$.

The above argument makes it clear that if in fact, for the specific $\mathcal{A}, \mathcal{B}$, there is a page that is read 4 times, the page must belong to the non-empty intersection $PgI(WT_{\mathcal{A}}) \cap PgII(WT_{\mathcal{B}}, \mathcal{B})$. Whenever this intersection becomes empty, Lemma 6 above can be strengthened to give back the three-reads per page guarantee. In any case, it should be noted that the guarantee provided is a very pessimistic worst-case. Experimental results reported in Section 5 will illustrate that in practice, most pages are only needed once or twice.

4.2 Grouping Multiple Nodes Satisfying a Path Relationship

So far, we have shown how to combine the processing of two nodes simultaneously, and we have presented an analysis to show how such grouped processing can be beneficial. Below, we generalize the grouping to multiple nodes and give corresponding analytic results.

Definition 5 Given the nodes/spaces $\mathcal{A}_1, \ldots, \mathcal{A}_w$, we say that these spaces satisfy a (directed) *path* relationship, denoted as $\langle \mathcal{A}_1, \ldots, \mathcal{A}_w \rangle$, if $\langle \mathcal{A}_i, \mathcal{A}_{i+1} \rangle$ satisfies an edge relationship for all $1 \leq i < w$. That is, there is a path in the lattice connecting all the nodes. □

For the example shown in Figure 3, the spaces $\{A, B, C, D\}$, $\{A, B, C\}$, $\{A, B\}$ and $\{A\}$ satisfy a path relationship. The following lemma generalizes the lemmas and corollaries given above for an edge relationship to a path relationship.

Lemma 7 Given the path relationship $\langle \mathcal{A}_1, \ldots, \mathcal{A}_w \rangle$:

- The combined set of Class I pages to process all w spaces simultaneously is given by: $PgI(WT_{\mathcal{A}_1} \cup \ldots \cup WT_{\mathcal{A}_w}) = PgI(WT_{\mathcal{A}_1})$.
- The combined set of Class II pages to process all w spaces simultaneously is given by: $\bigcup_{i=1}^{w} PgII(WT_{\mathcal{A}_i}, \mathcal{A}_i) = PgII(WT_{\mathcal{A}_w}, \mathcal{A}_w) \bigcup_{i=1}^{w-1} PgII((WT_{\mathcal{A}_i} - WT_{\mathcal{A}_{i+1}} - \ldots - WT_{\mathcal{A}_w}), \mathcal{A}_i)$.
- Compared with processing all w spaces separately, grouped processing of $\langle \mathcal{A}_1, \ldots, \mathcal{A}_w \rangle$ saves at least $(w-1)M + 2\sum_{i=2}^{w} M_{1,\mathcal{A}_i}$ page reads.
- When applying CELL to w spaces simultaneously, each page is guaranteed to be read ≤ 4 times. □

There are two interesting comments to make about the last two parts of the above lemma. First, with reference to Lemma 6, the last part says that regardless of how many edges there are on a path, the same worst-case guarantee of at most four-reads per page still applies. This suggests that the longer the path, the more shared I/O is possible and the larger is the saving. Second, the hidden overhead, however, is that the intersection between Class I pages and Class II pages grows as the length of the path increases. In other words, more and more pages are needed four times, instead of three.

4.3 Grouping Multiple Nodes Satisfying a Semi-Lattice Relationship

The generality of the analytic framework presented in Section 4.1 does not stop at the level of path relationships. We now show how to group and process even more nodes together, while preserving the properties and the I/O guarantee shown above.

Definition 6 Given multiple spaces $\mathcal{A}_1, \ldots, \mathcal{A}_w$, we say that these spaces satisfy a *semi-lattice* relationship, if there exists a $\top$-element $\mathcal{A}_i$ for some $1 \leq i \leq w$ such that the entire set $\{\mathcal{A}_1, \ldots, \mathcal{A}_w\}$ consists of all the subspaces of $\mathcal{A}_i$ and nothing else. □

For the example shown in Figure 3, the spaces $\{A, B, C\}$, $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, $\{A\}$, $\{B\}$ and $\{C\}$ satisfy a semi-lattice relationship, with the $\top$-element being the space $\{A, B, C\}$.

Like Lemma 7 is for a path relationship, we can give a general lemma for a semi-lattice relationship. However, we decide to omit the general lemma here because the notation would be too messy for the points we want to make. Instead, we only give results for an instance of a semi-lattice relationship, and point out a few interesting aspects.

Lemma 8 Consider a semi-lattice relationship, with the $\top$-element being the space $\{A, B, C\}$.

- The combined set of Class I pages to process all 7 spaces simultaneously is given by: $PgI(WT_{ABC})$.
- The combined set of Class II pages to process all 7 spaces simultaneously is given by: $PgII(WT_A, A) \cup PgII(WT_B, B) \cup PgII(WT_C, C) \cup PgII(WT_{AB} - WT_A - WT_B, AB) \cup PgII(WT_{AC} - WT_A - WT_C, AC) \cup PgII(WT_{BC} - WT_B - WT_C, BC) \cup PgII((WT_{ABC} - WT_{AB} - \ldots - WT_A - \ldots), ABC)$.
- Compared to processing all 7 spaces separately, semi-lattice grouped processing saves at least $6M + 2M_{1,A} + 2M_{1,B} + 2M_{1,C} + 2M_{1,AB} + 2M_{1,BC} + 2M_{1,AC}$ page reads.
- When applying Algorithm CELL to the 7 spaces simultaneously, each page is still guaranteed to be read at most 4 times. □

It is interesting to note that the "at most four-reads per page" guarantee applies to the path relationship $\langle\{A, B, C\}, \{A, B\}, \{A\}\rangle$, *as well as to* the semi-lattice relationship with $\{A, B, C\}$ being the $\top$-element. The difference, again, is that there are many more pages that need to be read 4 times in the latter relationship.

Finally, it is conceivable that we can simultaneously process multiple semi-lattice relationships. This begs the question: In terms of grouped processing of multiple nodes, where is the limit?

- First, there is the limit on the memory side. Our analysis of grouped processing focuses on I/O, and necessitates that there be sufficient memory space to simultaneously handle multiple nodes. For very large datasets returning many outliers, multiple instances of semi-lattice processing may indeed be problematic.
- Second, there are in fact scenarios where grouped processing of multiple nodes may not be beneficial. To illustrate, this is a good point to tie our discussion back to Procedures JumpLattice and DrillDown shown in Figure 4. Consider the space $\{A, B, C\}$. If *no* outlier is found in $\{A, B, C\}$, DrillDown is never called from JumpLattice, and none of the subspaces needs to be examined. In contrast, if we begin by processing the whole semi-lattice relationship with $\{A, B, C\}$ as the $\top$-element, we will have done more CPU operations and I/O reads (four-reads versus three-reads) than necessary. In Section 5, we will compare these different strategies experimentally.

4.4 Grouped Processing with Algorithm NL

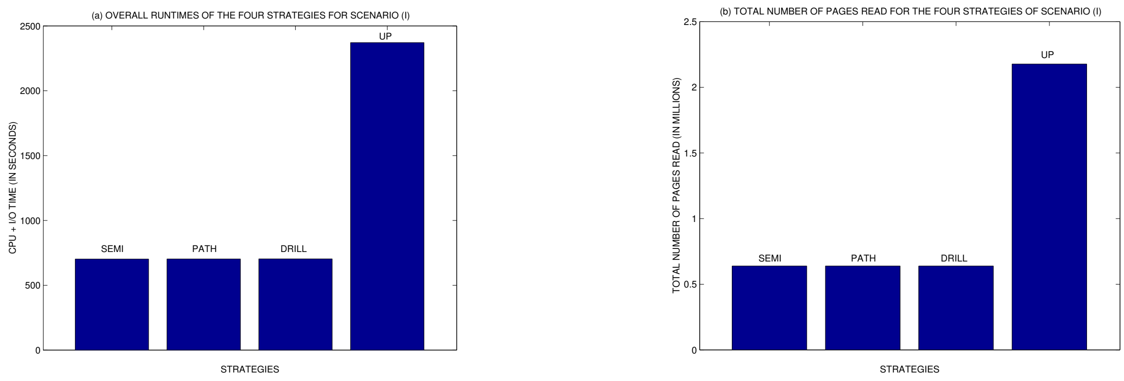
So far in this section, we have focused on grouped processing of multiple nodes using Algorithm CELL. This has been the focus because grouped processing is expected to be prevalent in situations where CELL is the baseline algorithm of choice and when strongest outliers are sought. But in some other situations, it is possible that Algorithm NL would need to be used as well (i.e., for k-D spaces where $k \geq 5$). Then, in those cases, as with CELL, grouped processing of nodes with NL can bring about an efficiency gain. For completeness, we outline how to do that with NL.

Recall that to allow CELL to process multiple nodes simultaneously, we deal with two key issues: (i) how to make the algorithm itself process multiple nodes/spaces (e.g., combined Class I and II pages), and (ii) how to group and select the multiple nodes effectively (e.g., path and semi-lattice relationships). Here, we examine the same two issues for NL. The first issue is conceptually simple. If there is only one space for NL to process, NL keeps for each tuple P a count of the tuples that are within the fixed-radius neighbourhood of P in that space. If there are w spaces for NL to process simultaneously, NL can keep for each tuple P a count for each of the w spaces. Because the w counts are so unrelated to each other (except that they share the same I/O), the second issue of grouping and selecting the w spaces for NL becomes simple as well, unlike the situation for CELL. Theoretically, any w spaces can do. In practice, though, the heuristic of picking the spaces in ascending cardinalities is the most sensible, assuming we wish to find the strongest outliers or the top-u strongest/weak outliers.

5 Experimental Evaluation

5.1 Experimental Setup

Our base dataset is an 855-record dataset consisting of 1995-96 NHL player performance statistics. These statistics include numbers of goals, assists, points, penalty minutes, shots on goal, etc. Since this real-life dataset is quite small, and since we want to test our algorithms on large, disk-resident datasets, we created a number of synthetic datasets mirroring the distribution of statistics within the NHL dataset. Specifically, we determined the distribution of the attributes in the original dataset by using a 10-partition histogram. Then, we generated datasets containing up to 2 million tuples, and whose distribution mirrored that of the base dataset. Each data page held up to 13 tuples.



(a) Overall Runtime

(b) Number of Pages Read

Figure 6: Scenario I: Outliers Found Only in 3-D Spaces

We implemented all four strategies analyzed so far. Specifically, for the results reported below:

- "UP" denotes UpLattice shown in Figure 4.
- "DRILL" denotes JumpLattice using DrillDown to process the subspaces if necessary.
- "PATH" denotes JumpLattice that, instead of calling DrillDown, applies grouped processing of nodes satisfying a path relationship.
- Finally, "SEMI" denotes JumpLattice that applies grouped processing of nodes satisfying a semi-lattice relationship.

Our tests were run on a time-sharing environment provided by a Sun Microsystems Ultra-60 workstation. Unless otherwise indicated, all times shown in this paper are CPU times plus I/O times. Buffer management was done by the operating system. Even though different buffer management strategies may change the reported results in absolute terms, they do not change how the various algorithms compare with one another in relative terms.

5.2 Computing All Strongest Outliers

In the first set of experiments, we focus on the computation of all strongest outliers. The results reported below are based on a 4-dimensional space. We compare the number of I/O operations and the overall runtime required by the four different strategies. As discussed before, the relative performance of the strategies depends on how many outliers are contained in the 3-D space, and in its 2-D and 1-D subspaces. Below, we report on the results of three different scenarios: (I) outliers found only in the 3-D space but not in any of the subspaces, (II) a small number of outliers (e.g., 10) found in some 2-D subspace(s) but not in any of the 1-D subspaces, and (III) a large number of outliers (e.g., 400) found in the 2-D subspaces. Later in this section, we comment on the scenario where all strongest outliers are found at the 1-D level.

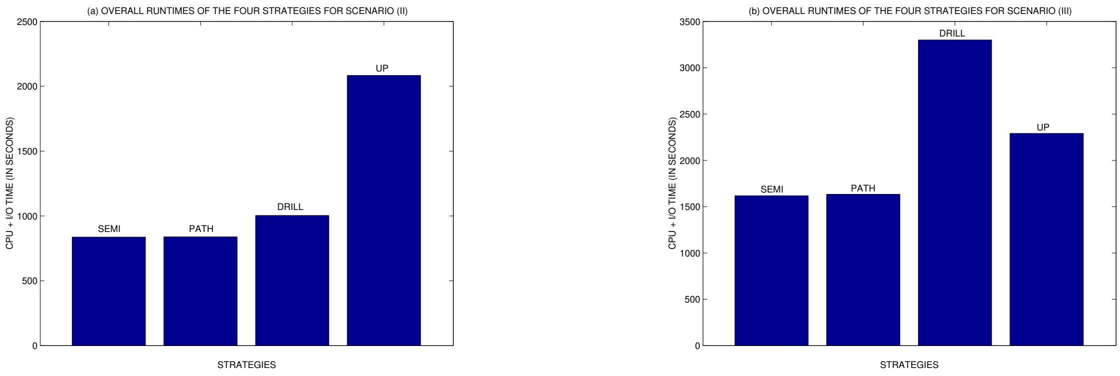
Figure 6 shows how the four strategies compare under Scenario (I). Figure 6(a) compares their overall runtimes, and Figure 6(b) shows their total number of pages read for 2 million tuples. UpLattice is the only strategy that does poorly, while the other three give almost identical performance running more than three times faster than, and requiring only about 30% of the I/O's needed by, UpLattice. Thus, Scenario (I) is highly unfavourable to the bottom-up approach taken by UpLattice. Note that there is a strong 1-to-1 correspondence between the overall runtime and the number of pages read, indicating that this is a very I/O dominant job. From now on, we only show the overall runtime figures.

Figures 7(a) and (b) show the overall runtimes of the four strategies under Scenarios (II) and (III), respectively. While UpLattice remains a poor strategy, DrillDown can do worse. Both the path and semi-lattice strategies continue to be best, with relatively little difference between them. Notice that as there are more outliers contained in the 2-D subspaces, the gap between UpLattice and the path and semi-lattice strategies becomes smaller. Eventually, when all 1-D subspaces contain outliers, even the path and semi-lattice strategies deteriorate beyond UpLattice. In practice though, the scenario where all strongest outliers are found in 1-D subspaces, is expected to be rare. For more typical scenarios, it is clear that the path and the semi-lattice strategies are the ones to use.

5.3 Computing Top-u Outliers

The experimental results reported so far pertain to strongest outliers only. In this set of experiments, we compare the performance of the four strategies for computing the top-u non-trivial (i.e., strongest or weak) outliers. Recall that when computing strongest outliers only, once a space is found to contain outliers, none of its superspaces needs to be computed. But when it is necessary to compute the weak outliers also, even the superspaces need to be examined. In general, this scenario favours the path and semi-lattice strategies because of the shared processing.

Figure 8 shows how the overall runtime changes for



(a) Scenario II (b) Scenario III

Figure 7: Outliers Found in 2-D Subspaces: Overall Runtime

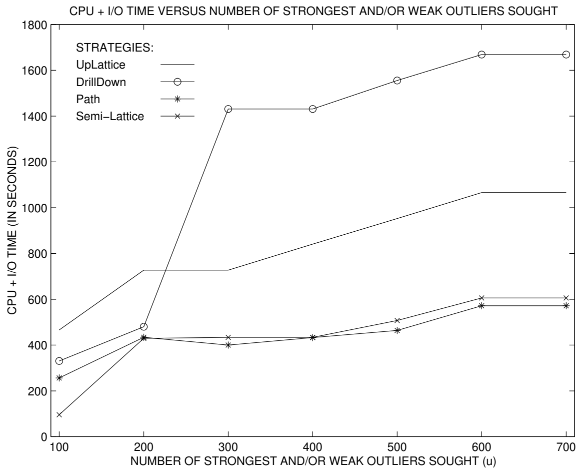


Figure 8: Top-u Strongest and/or Weak Outliers

computing top-u outliers when u varies from 100 to 700, for 500,000 tuples. UpLattice starts off poorly even for small values of u, and remains poor as u increases. DrillDown starts off very well, but quickly degrades to be the worst. Both the path and the semi-lattice strategies are again the winners, with a slight edge being given to the path strategy. The main reason for this difference is: because computation stops once the first u non-trivial outliers are found, the semi-lattice strategy may over-do what is actually needed.

With respect to the four-reads per page guarantee, we provide the following results to give readers an appreciation for the fact that four-reads per page are encountered relatively infrequently in practice. For example, for a 2 million tuple dataset using the semi-lattice strategy, with 4.8 million pages read in all, 0.17 million pages were read exactly once, 1.7 million page were read exactly twice, 0.4 million pages were read exactly three times, and only 13,025 pages were read exactly four times.

In Section 4.3, we pointed out that we can generalize from the path strategy to the semi-lattice strategy, and even further. This leads to the question of how far we can go and where we should stop. The experimental results presented so far indicate that for computing strongest/weak outliers, there appears to be little promise to go beyond the semi-lattice strategy. In fact, between the path and the semi-lattice strategies, we recommend the former. It gives results at least as good as the semi-lattice strategy, and it is simpler in many ways.

5.4 Grouped Processing with Algorithm NL

So far, we have seen how grouped processing pays off in lower dimensional spaces using Algorithm CELL. We now turn our attention to higher dimensional spaces. As mentioned before, Algorithm CELL is the preferred algorithm for $k \leq 4$. Now suppose that a lattice contains two or more unprocessed nodes of cardinality ≥ 5, that more outliers are sought, and that all processing for nodes of cardinality ≤ 4 is complete. Then, we can either use Algorithm NL to process each remaining node on an individual basis, or we can group nodes and thus share I/O's.

Below are some performance results for grouped NL processing, in which seven nodes are processed together—one $k = 6$ node and all six of its $k = 5$ subspaces. For a dataset containing 500,000 tuples, using 25% buffering, individual NL processing on these seven nodes took 24% longer than with grouped NL processing. For this case, grouped processing requires 16 MB of memory, which is almost twice as much memory as each individual case requires. If we were to equate the memory allocations to about 8 MB (by providing only 9% buffering for the grouped case), then individual processing took only about 1% more time than grouped processing. Although we do not make performance guarantees, we observe that grouped NL processing generally yields only modest gains (e.g., 10%). This is due to the fact that the overall time for grouped NL processing is roughly broken down into 95% CPU time and 5% I/O time. Thus, while it is true that I/O operations are reduced significantly, the impact of shared I/O may not be significant overall. In summary, because of simplicity and the somewhat unpredictable nature of finding outlying spaces, we conclude that individual NL processing should be used in place of grouped NL processing.

6 Conclusions

This paper focused on finding intensional knowledge of outliers to help explain why the identified outliers are exceptional. To the best of our knowledge, we are the first to go beyond identification to explanation.

The intensional knowledge considered in this paper is structural in nature. Through the concepts of strongest and weak outliers, our algorithms report, for each identified outlier P in the original attribute space $\mathcal{A}_P$, the minimal subspaces in which P is outlying, and how P compares against other identified outliers with respect to these minimal subspaces. We believe that the reported intensional knowledge is valuable to the user in helping to evaluate the validity of the identification, and in improving the user's understanding of the dataset in general and the outliers in particular.

We presented four strategies for the computation of strongest and weak outliers: UpLattice, JumpLattice with DrillDown, JumpLattice with Path, and JumpLattice with Semi-Lattice. For the latter two strategies, we showed that the kinds of grouped processing they perform are correct, and we presented a detailed analysis on their I/O performance. We provided two I/O performance guarantees: one on the minimum number of I/O's saved, and the other on the maximum number of times a data page has to be read.

The I/O analytic results are confirmed by experimental results. Under various scenarios, grouped processing with the path or the semi-lattice strategies can bring about a 65–75% reduction in I/O operations and overall runtime, when compared with the alternatives. And even though computing intensional knowledge appears to be time-consuming, especially for large and/or high dimensional datasets, our results clearly show that effective sharing of I/O's, via the path or the semi-lattice strategies, can go a long way towards reducing the overall runtime.

References

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications," *Proc. ACM SIGMOD*, pp. 94–105, 1998.

[AGI+92] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. "An Interval Classifier for Database Mining Applications", *Proc. VLDB*, pp. 560–573, 1992.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. "Mining Association Rules between Sets of Items in Large Databases", *Proc. ACM SIGMOD*, pp. 207–216, 1993.

[BL94] V. Barnett and T. Lewis. *Outliers in Statistical Data.* John Wiley & Sons, 1994.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth International Group, 1984.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. "Beyond Market Basket: Generalizing Association Rules to Correlations", *Proc. ACM SIGMOD*, pp. 265–276, 1997.

[HKPT98] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. "Efficient Discovery of Functional and Approximate Dependencies Using Partitions", *Proc. ICDE*, 1998.

[JKN98] T. Johnson, I. Kwok, and R. Ng. "Fast Computation of 2-Dimensional Depth Contours," *Proc. KDD*, pp. 224–228, 1998.

[KR90] L. Kaufman and P. Rousseeuw. *Finding Groups in Data.* John Wiley & Sons, 1990.

[KN98] E. Knorr and R. Ng. "Algorithms for Mining Distance-Based Outliers in Large Datasets," *Proc. VLDB*, pp. 392–403, 1998.

[KN99] E. Knorr and R. Ng. "Finding Intensional Knowledge of Distance-Based Outliers," Technical Report, Dept. of Computer Science, University of British Columbia.

[NLHP98] R. Ng, L. Lakshmanan, J. Han, and A. Pang. "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules", *Proc. ACM SIGMOD*, pp. 13–24, 1998.

[RR96] I. Ruts and P. Rousseeuw. "Computing Depth Contours of Bivariate Point Clouds", *Computational Statistics and Data Analysis*, 23, pp. 153–168, 1996.

[SBMU98] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. "Scalable Techniques for Mining Causal Structures", *Proc. VLDB*, pp. 594–605, 1998.

[TN98] V. Tucakov and R. Ng. "Identifying Unusual People Behavior: A Case Study of Mining Outliers in Spatio-Temporal Trajectory Databases", *Proc. SIGMOD Workshop on Research Issues on Knowledge Discovery and Data Mining*, 1998.

[Tuk77] J. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.