Efficient Resumption of Interrupted Warehouse Loads*

Wilburt Juan Labio†, Janet L. Wiener‡, Hector Garcia-Molina
Stanford University
{wilburt, wiener, hector}@db.stanford.edu

Vlad Gorelik
Sagent Technologies
vgorelik@sagenttech.com

Abstract

Data warehouses collect large quantities of data from distributed sources into a single repository. A typical load to create or maintain a warehouse processes GBs of data, takes hours or even days to execute, and involves many complex and user-defined transformations of the data (e.g., find duplicates, resolve data inconsistencies, and add unique keys). If the load fails, a possible approach is to "redo" the entire load. A better approach is to resume the incomplete load from where it was interrupted. Unfortunately, traditional algorithms for resuming the load either impose unacceptable overhead during normal operation, or rely on the specifics of transformations. We develop a resumption algorithm called *DR* that imposes no overhead and relies only on the high-level properties of the transformations. We show that *DR* can lead to a ten-fold reduction in resumption time by performing experiments using commercial software.

1 Introduction

Data warehouses collect large quantities of data from distributed sources into a single repository. A typical load to create or maintain a warehouse processes 1 to 100 GB and takes hours to execute. For example, Walmart's maintenance load averages 16 GB per day [3]. Typical maintenance loads of Sagent customers process 6 GB per week and initial loads process up to 100 GB.

Warehouse loads are usually performed when the system is off-line (e.g., overnight), and must be completed within a fixed period. A failure during the load creates havoc: Current commercial systems abort the failed load, and the administrator must restart the load from scratch and hope a second failure does not occur. If there is not enough time for the new load, it may be skipped, leaving the database out of date or incomplete, and generating an even bigger load for the next period. Load failures are not unlikely due to the complexity of the warehouse load. For instance, Sagent customers report that one out of every thirty loads fails [10].

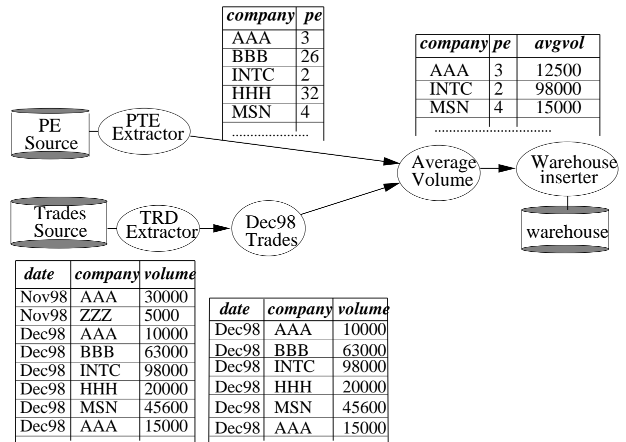

Figure 1: Load Workflow

Traditional recovery techniques described below could be used to save partial load states, so that not all work is lost when a failure occurs. However, these techniques are shunned in practice because they generate high overheads during normal processing and because they may require modification of the load processing. In this paper we present a new, low-overhead technique for *resuming* failed loads. Our technique exploits high-level "properties" of the workflow used to load the warehouse, so that work is not repeated during a resumed load.

To illustrate the type of processing performed during a load, consider the simple load workflow of Figure 1. In this load workflow, *extractors* obtain data from the stock Trades and the price-to-earnings ratio (PE) sources. Figure 1 shows a sample prefix of the tuples extracted from each source. The Trades data is first processed by the *Dec98Trades transform*,

---

*This research was funded by Rome Laboratories under Air Force Contract F30602-94-C-0237, by the Massive Digital Data Systems (MDDS) Program sponsored by the Advanced Research and Development Committee of the Community Management Staff, and by Sagent Technologies, Inc.

†Currently at Gigabeat, Inc. Palo Alto CA; wilburt@gigabeat.com

‡Currently at Compaq SRC, Palo Alto, CA; wiener@pa.dec.com.

Permission to make digital or hard copies of part or all of this work or 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.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

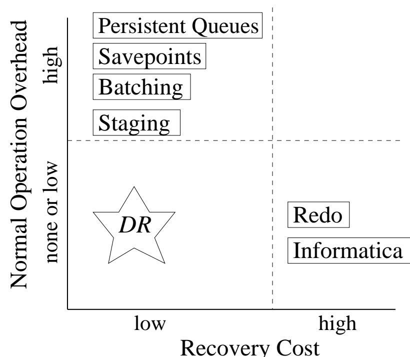

Figure 2: Applicability of Algorithms

which only outputs trades done in December 1998. Thus, the first two trades are removed since they happened in November 1998. Then, the *AverageVolume* transform groups the trades by company and finds the average trade volume of the companies whose pe is less than or equal to 4. For instance, companies BBB and HHH are discarded since they have high pe's. The output of *AverageVolume* is then sent to the *inserter*, which stores the tuples in the warehouse.

In practice, load workflows can be much more complex than what we have illustrated, often having tens to hundreds of transforms [10]. Transforms include not only conventional database operations (e.g., join) but also arbitrary processing (e.g., data scrubbing, byte reordering) coded by application specialists. To maximize pipelining and load speed, the output tuples of each component are sent to the next as soon as they are generated. With all this complexity, load failures occur.

There are many ways to recover a failed warehouse load. The fundamental features of various techniques are informally contrasted with our technique, called *DR*, in Figure 2. The vertical axis represents the *normal-operation overhead* of a technique, while the horizontal axis indicates the *recovery cost* of a technique. In the lower right quadrant of Figure 2 are two techniques that have very low normal-operation overhead. One is to simply redo the entire load over again. Clearly, this technique can suffer from high recovery cost. Informatica's solution [6] is similar: After a failure, Informatica reprocesses all the data, but filters out already stored tuples when they reach the warehouse for the second time (i.e., just before the inserter).

Other techniques, shown in the upper left quadrant of Figure 2, attempt to minimize the recovery cost by aggressively modifying the load workflow or load processing. Staging divides the workflow into consecutive stages, and saves intermediate results between stages. All input data enters the first stage. All of the first stage's output is saved. The saved output is input to the second stage, and so on. The second stage can then be restarted after a failure from the saved input, without redoing the first stage.

Input batching divides the input to the load workflow into sequentially processed batches. Other techniques are to take periodic savepoints [5] of the workflow state, or save tuples in transit in persistent queues [1,2]. When a failure occurs, the *modified* transforms cooperate to revert to the latest savepoint, and proceed from there.

In general, techniques that modify the load workflow suffer from two disadvantages: (1) the normal-operation overhead is potentially high, as confirmed by our experiments; and (2) the specific details of the load processing need to be known.

The *DR* technique we propose in this paper has no normal-operation overhead, and does not modify the load workflow. Yet, its recovery cost can be much lower than Informatica's technique or redoing the entire load. Unlike them, *DR* avoids reprocessing input tuples and uses filters to intercept tuples much earlier than Informatica's technique. *DR* relies on simple and high-level transform properties (e.g., are tuples processed in order?). These properties can either be declared by the transform writer or can usually be inferred from the basic semantics of the transform, without needing to know exactly how it is coded. After a failure, the load is restarted, except that portions that are no longer needed are "skipped." To illustrate, suppose that after a failure we discover that tuples AAA through MSN are found in the warehouse (Figure 1. If we know that tuples are processed in alphabetical order by the *PTE Extractor* and by the *AverageVolume* transform, the *PTE Extractor* can retrieve tuples starting with the one that follows MSN. If tuples are not processed in order, it may still be possible to generate a list of company names that are no longer needed, and that can be skipped. During the reload, transforms operate as usual, except that they only receive the input tuples needed to generate what is missing in the warehouse. In summary, our strategy is to exploit some high-level semantics of the load workflow, and to be selective when resuming a failed load.

We note that there are previous techniques that are similar to *DR* in that they incur low normal-operation overhead but still have a low recovery cost. However, these techniques are applicable to very specific workflows for disk-based sorting [8], object database loading [11], and loading a flat file into the warehouse [9,12]. Our technique can handle more general workflows.

We do not claim that *DR* always recovers a load faster than other techniques. For instance, techniques that modify the normal load workflow may have slower normal operation loads but faster resumed loads. However, our experiments show that *DR* is competitive if not better than these techniques for many workflows. In particular, *DR* is better for workflows that make heavy use of pipelining. Even if a workflow does not have a natural pipeline, our experiments show that a hybrid algorithm that combines *DR* and staging (or batching) can lower recovery cost.

We make the following contributions toward the efficient resumption of failed warehouse loads.

- We develop a framework for describing successful warehouse loads, and load failures. Within this framework, we identify basic properties that are useful in resuming loads.

- We develop *DR*, which minimizes recovery cost while imposing no overhead during normal operation. *DR* does not require knowing the specifics of a transform, but only its basic, high-level properties. *DR* is presented here in the context of warehousing, but is really a generic solution for resuming any long-duration, process intensive, task.
- We show experimentally that *DR* can significantly reduce recovery cost, as compared to traditional techniques. In our experiments we use Sagent's warehouse load package to load TPC-D tables and materialized views containing answers to TPC-D queries.

Outline: We describe a warehouse load in Section 2, and discuss warehouse load failure in Section 3. We develop the *DR* algorithm in Sections 4 and 5. Experiments are presented in Section 6.

2 Normal Operation

When data is loaded into the warehouse, tuples are transferred from one *component* (*extractor*, *transform*, or *inserter*) to another. The order of the tuples is important to the resumption algorithm, so we define sequences as ordered lists of tuples with the same attributes.

Definition 2.1 (**Sequence**) A sequence of tuples $\mathcal{T}$ is an ordered list of tuples $[t_1..t_n]$, and all the tuples in $\mathcal{T}$ have the attributes $[a_1..a_m]$. □

We next discuss how a component tree represents a load workflow. In [7], we show how our *DR* algorithm can be extended to handle a component directed acyclic graph.

2.1 Component Tree Design

Figure 3 illustrates the same component tree as Figure 1, with abbreviations for the component names. Constructing a component tree involves several important design decisions. First, the data obtained by the extractors is specified. Second, the transforms that process the extracted data are chosen. Moreover, if a desired transformation is not available, a user may construct a new custom-made transform. Finally, the warehouse tables into which the inserter loads the data are specified. The extractors, transforms, and inserter comprise the *nodes* of the component tree.

Each transform and inserter expects certain *input parameter* sequences at load time. Similarly, each transform and extractor generates an output sequence to its *output parameter*. The input and output parameters are specified by connecting the extractors, transforms, and the inserter together with *edges* in the component tree. In the design, the "properties" that hold for each node or edge are declared for use by our resumption algorithm, as detailed in Section 4. Commercial load packages already declare basic properties like the key attributes of an input parameter.

In some cases, different components of a tree may be assigned to different machines. Hence, during a load, data transfers between components may represent data transfers over the network. We now illustrate a component tree.

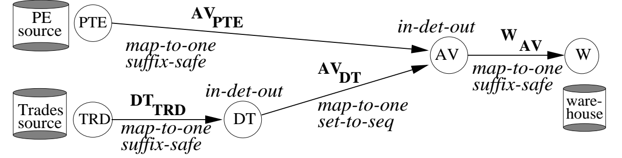

Figure 3: Component Tree with Properties

Example 2.1 In Figure 3, the extractors are denoted PTE for the price-to-earnings (PE) source, and TRD for the Trades source. The transforms are denoted DT (for $Dec98$-$Trades$), and AV (for $AverageVolume$). The inserter is denoted W. The input parameters of each component are denoted by the component that produces the input. For instance, AV_{DT} is an input parameter of AV that is produced by DT. □

In our notation, Y_X denotes the input parameter of component Y produced by component X, and Y_O is the output parameter of Y. We use Attrs(Y_X) to denote the attributes of the Y_X tuples. Similarly, KeyAttrs(Y_X) specifies their key attributes. W denotes the warehouse inserter.

We note that the component trees designed for warehouse creation and maintenance are different [7]. However, our resumption algorithm applies equally well to both creation and maintenance component trees. We therefore use the term "load" to mean either one.

2.2 Successful Warehouse Load

When a component tree is used to load data, the extractors produce sequences that are inputs to the transforms. That is, each input parameter is "instantiated" with a tuple sequence. Each transform then produces an output sequence that is sent to subsequent components. Finally, the inserter receives a tuple sequence and inserts the tuples in committed batches. To maximize pipelined parallelism, each output sequence is received as the next component's input as it is generated. More specifically, at each point in time, a component Y has produced a prefix of its entire output sequence and shipped the prefix tuples to the next component. The next example illustrates a warehouse load during normal operation, i.e., no failures occur.

Example 2.2 Consider Figure 3. The extractors fill their output parameters PTE_O and TRD_O with the sequences $\mathcal{PTE}_O$ and $\mathcal{TRD}_O$. (The calligraphy font denotes sequences.) Input parameter AV_{PTE} is instantiated with the sequence $\mathcal{AV}_{PTE} = \mathcal{PTE}_O$. Note that PTE does not need to produce $\mathcal{PTE}_O$ in its entirety before it can ship a prefix of $\mathcal{PTE}_O$ to AV. Similarly, DT_{TRD} is instantiated with $\mathcal{DT}_{TRD} = \mathcal{TRD}_O$, and so on. Finally, W_{AV} of the inserter is instantiated with $\mathcal{W}_{AV} = \mathcal{AV}_O$. W inserts the tuples in $\mathcal{W}_{AV}$ in order and issues a commit periodically. In the absence of failures, $\mathcal{W}_{AV}$ is eventually stored in the warehouse. □

To summarize our notation, $\mathcal{Y}_X$ and $\mathcal{Y}_O$ denote the sequences used for input parameter Y_X and output parameter Y_O during a warehouse load. When Y produces $\mathcal{Y}_O$ by processing $\mathcal{Y}_X$ (and possibly other input sequences), we say $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$. $\mathcal{W}$ denotes the sequence that is loaded into the warehouse in the absence of failures.

3 Warehouse Load Failure

In this paper, we only consider system-level load failures, e.g., RDBMS or software crashes, hardware crashes, lack of disk space. We do not consider load failures due to invalid data. Furthermore, we consider system-level failures that do not affect information stored in stable storage. Any of the components can suffer from such a system-level failure. Even though various components may fail, the effect of any failure on the warehouse is the same. That is, only a prefix of the normal operation input sequence $\mathcal{W}$ is loaded into the warehouse.

Observation In the event of a failure, only a prefix of $\mathcal{W}$ is stored in the warehouse. □

In [7], we discuss in detail why the observation holds when an extractor, a transform, an inserter or the network fails.

3.1 Data for Resumption

When a component Y fails, the warehouse load eventually halts due to lack of input. Once Y recovers, the load can be resumed. The only data available for resumption is the portion of $\mathcal{W}$ in the warehouse, plus the source data. An extractor E may offer some of the following procedures to re-extract data. We use $\mathcal{E}_O$ to denote the sequence that would have been extracted by E had there been no failures. More details are in Section 5.3.

GetAll() extracts the same set of tuples as are in $\mathcal{E}_O$. The order of the tuples may be different. (Many sources, such as commercial RDBMS, do not guarantee order.) We assume that all extractors provide GetAll(), that is, that the original data is still available. If $\mathcal{E}_O$ cannot be reproduced, then $\mathcal{E}_O$ must be logged.

GetAllInorder() extracts the same sequence $\mathcal{E}_O$. This procedure may be supported, e.g., by a commercial RDBMS extractor that uses an SQL ORDER BY clause.

GetSubset(...) provides the $\mathcal{E}_O$ tuples that are not in the subset indicated by GetSubset's parameters. Sources that can selectively filter tuples typically provide GetSubset.

GetSuffix(...) provides a suffix of $\mathcal{E}_O$ that excludes the prefix indicated by GetSuffix's parameters. Sources that can filter and order tuples typically provide GetSuffix.

In this paper, we assume that the re-extraction procedures only produce tuples that were in the original sequence $\mathcal{E}_O$. However, our algorithms also work when additional tuples appear only in the suffix of $\mathcal{E}_O$ that was not processed before the failure.

3.2 Redoing the Warehouse Load

When the warehouse load fails, only a prefix $\mathcal{C}$ of $\mathcal{W}$ is in the warehouse. The goal of a resumption algorithm is to load the remaining tuples of $\mathcal{W}$ in any order (since the warehouse is an RDBMS). The simplest resumption algorithm, called *Redo*, simply repeats the load. First $\mathcal{C}$ is deleted, and then for each extractor in the component tree, the re-extraction procedure GetAll() is invoked.

Although *Redo* is very simple, it still requires that if the same set of tuples are obtained by the extractors, the same set of tuples are inserted into the warehouse. Since this property pertains to an entire workflow, it can be hard to test. A *singular property* that pertains to a single transform is much easier to test. The following singular property, set-to-set, is sufficient to enable *Redo*. That is, if all extractors use GetAll or GetAllInorder, and all transforms are set-to-set, then *Redo* can be used. Definition 3.1 tests this condition.

Property 3.1 (**set-to-set**(Y)) If given the same set of input tuples, Y produces the same set of output tuples, then set-to-set(Y) = true. Otherwise, set-to-set(Y) = false. □

Definition 3.1 (**Same-set**(Y)) If Y is an extractor and Y uses GetAllInorder or GetAll during resumption then Same-set(Y) = true. Otherwise, if $\forall Y_X$: Same-set(X) and set-to-set(Y) then Same-set(Y) = true. Otherwise, Same-set(Y) = false. □

4 Properties for Resumption

In this section, we identify singular properties of transforms or input parameters that *DR* combines into "transitive properties" to avoid reprocessing some of the input tuples.

To illustrate, consider the simple component tree in Figure 4. Suppose that the sequence $\mathcal{W}_Y$ to be inserted into the warehouse is $[y_1 y_2 y_3]$, and $[x_1 x_2 x_3 x_4]$ is the $\mathcal{Y}_X$ input sequence that yields $\mathcal{W}_Y$ (Figure 5). An edge $x_i \rightarrow y_j$ indicates that x_i "contributes" in the computation of y_j. (We define contributes formally in Definition 4.1.) Also suppose that after a failure, only y_1 is in the warehouse. Clearly, it is safe to filter $\mathcal{Y}_X$ tuples that contribute only to $\mathcal{W}_Y$ tuples, such as y_1, already in the warehouse. Thus in Figure 5, x_1 and x_2 can be filtered. We need to be careful with y_1 contributors that also contribute to other $\mathcal{W}_Y$ tuples. For example, if x_2 contributes to y_2 as well, then we cannot filter x_2, since it is still needed to generate y_2.

In general, we need to answer the following questions to avoid reprocessing input tuples:

Question (1): For a given warehouse tuple, which tuples in $\mathcal{Y}_X$ contribute to it?

Question (2): When is it safe to filter those tuples from $\mathcal{Y}_X$?

The challenge is that we must answer these questions using limited information. In particular, we can only use the tuples stored in the warehouse before the failure, and the singular properties, attributes and key attributes declared when the component tree was designed.

In Section 4.1, we identify four singular properties to answer Question (2). We then define three *transitive properties* that apply to sub-trees of the component tree. *DR* will

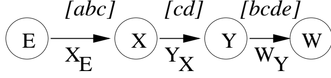

Figure 4: Component tree

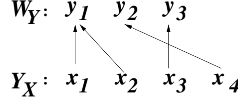

Figure 5: Map-to-one

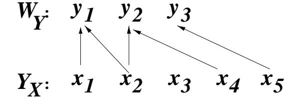

Figure 6: Suffix-safe

derive the transitive properties based on the declared singular properties. In Section 4.2, we define two more singular properties. Using these properties, we define *identifying attributes* of the tuples to answer Question (1). *DR* will derive the identifying attributes based on the declared singular properties and key attributes. We also show that the singular properties hold for many commercial transforms, such those in Sagent's DataMart 3.0 for warehouse creation and maintenance, in Section 6. Since singular properties pertain to a transform or an input parameter and not to a whole workflow, they are easy to grasp and can often be deduced easily from the transform manuals. Henceforth, we refer to singular properties as "properties" for conciseness.

Before proceeding, we formalize the notion of contributing input tuples. An input tuple x_i in an input sequence $\mathcal{Y}_X$ of transform Y *contributes* to a tuple y_j in a resulting output sequence $\mathcal{Y}_O$ if y_j is only produced when x_i is in $\mathcal{Y}_X$. The definition of "contributes" uses the function IsSubsequence($\mathcal{S}$, $\mathcal{T}$), which returns true if $\mathcal{S}$ is a subsequence of $\mathcal{T}$, and false otherwise.

Definition 4.1 (**Contributes, Contributors**) Given transform Y, let $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$ and $Y(...\mathcal{Y}'_X...) = \mathcal{Y}'_O$. Also let $\mathcal{Y}_X = [x_1..x_{i-1}x_ix_{i+1}..x_n]$ and $\mathcal{Y}'_X = [x_1..x_{i-1}x_{i+1}..x_n]$. Contributes($x_i$,$y_j$) = true, if $y_j \in \mathcal{Y}_O$ and $y_j \notin \mathcal{Y}'_O$. Otherwise, Contributes(x_i,y_j) = false. Contributors($\mathcal{Y}_X$, y_j) = $\mathcal{T}$, where IsSubsequence($\mathcal{T}$, $\mathcal{Y}_X$) and ($\forall x_i \in \mathcal{T}$: Contributes(x_i, y_j)) and ($\forall x_i \in \mathcal{Y}_X$: Contributes(x_i,y_j) $\Rightarrow x_i \in \mathcal{T}$). □

We extend Definition 4.1 in a transitive fashion in [7] to define when a tuple contributes to a warehouse tuple. For instance, if x_i contributes to y_j, which in turn contributes to a warehouse tuple w_k, then x_i contributes to w_k.

Some tuples may not contribute to any output tuple. For instance, if a transform computes the sum of its input tuples then an input tuple $t = \langle 0 \rangle$ does not contribute to the sum. Such tuples are called *inconsequential input tuples* and are candidates for filtering.

4.1 Safe Filtering

During resumption, a transform Y may not need to produce all of its normal operation output $\mathcal{Y}_O$. Therefore, Y may not need to reprocess some of its input tuples. In this section, we identify properties that ensure safe filtering of input tuples.

The *map-to-one* property holds for Y_X whenever every input tuple x_i contributes to at most one Y_O output tuple y_j (as in Figure 5). For instance, the input parameters of selection, projection, union, aggregation and some join transforms are map-to-one. Nearly all of Sagent's input parameters are map-to-one.

Property 4.1 (**map-to-one**(Y_X)) Given transform Y with input parameter Y_X, Y_X is *map-to-one* if $\forall \mathcal{Y}_X, \forall \mathcal{Y}_O, \forall x_i \in \mathcal{Y}_X$: $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow (\neg\exists y_j, y_k \in \mathcal{Y}_O$ such that Contributes(x_i,y_j) and Contributes(x_i,y_k) and $j \neq k$). □

If Y_X is map-to-one, and some tuples in $\mathcal{Y}_O$ are not needed, then their contributing tuples in $\mathcal{Y}_X$ can be safely filtered at resumption time. For example, in Figure 5, if tuples y_1 and y_2 are not needed in $\mathcal{Y}_O$, then the subset $\{x_1, x_2, x_4\}$ of Y_X can be filtered.

Subset-feasible(Y_X) is a transitive property that states that it is feasible to filter some subset of the Y_X input tuples. Subset-feasible(Y_X) holds when all of the input parameters in the path from Y_X to the warehouse are map-to-one. In this case, we can safely filter the Y_X tuples that contribute to some warehouse tuple because these Y_X tuples contribute to no other warehouse tuples.

Definition 4.2 (**Subset-feasible**(Y_X)) Given transform Y with input parameter Y_X, Subset-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Subset-feasible(Y_X) = true if Y_X is map-to-one and Subset-feasible(Z_Y). Otherwise, Subset-feasible(Y_X) = false. □

While the map-to-one and subset-feasible properties allow a *subset* of the input sequence to be filtered, the *suffix-safe* property allows a *prefix* of the input sequence to be filtered. The suffix-safe property holds when any prefix of the output can be produced by some prefix of the input sequence. Moreover, any suffix of the output can be produced from some suffix of the input sequence. For instance, the input parameters of transforms that perform selection, projection, union, and aggregation over sorted input are suffix-safe.

Property 4.2 (**suffix-safe**(Y_X)) Given $\mathcal{T} = [t_1..t_n]$, let First($\mathcal{T}$) = t_1, Last($\mathcal{T}$) = t_n, and $t_i \leq_{\mathcal{T}} t_j$ if t_i is before t_j in $\mathcal{T}$ or $i = j$. Given transform Y with input parameter Y_X, Y_X is *suffix-safe* if $\forall \mathcal{Y}_X, \forall \mathcal{Y}_O, \forall y_j, y_{j+1} \in \mathcal{Y}_O$: $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow$ (Last(Contributors($\mathcal{Y}_X$, y_j)) $\leq_{\mathcal{Y}_X}$ First(Contributors($\mathcal{Y}_X$, y_{j+1}))). □

Figure 6 illustrates conceptually how suffix-safe can be used. If only $[y_3]$ of $\mathcal{Y}_O$ in Figure 6 needs to be produced, processing the suffix $[x_5]$ of $\mathcal{Y}_X$ will produce $[y_3]$. Conversely, if $[y_1y_2]$ does not need to be produced, the prefix $[x_1x_2x_3x_4]$ can be filtered from Y_X at resumption time. Notice that when the suffix-safe property is used, inconsequential input tuples like x_3 can be filtered. Filtering such tuples is not possible using the map-to-one property. 78% of Sagent's transforms' input parameters are suffix-safe.

Prefix-feasible(Y_X) is a transitive property that states that it is feasible to filter some prefix of the Y_X input sequence. This property is true if all of the input parameters from Y_X to the warehouse are suffix-safe. (The reasoning is similar to that for Subset-feasible(Y_X) and map-to-one.)

Definition 4.3 (**Prefix-feasible(Y_X)**) Given transform Y with input parameter Y_X, Prefix-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Prefix-feasible(Y_X) = true if Y_X is suffix-safe and Prefix-feasible(Z_Y). Otherwise, Prefix-feasible(Y_X) = false. □

Filtering a prefix of the Y_X input sequence is possible only if Y_X receives the same sequence during load resumption as it did during normal operation. For instance, in Figure 6, even if Prefix-feasible(Y_X) holds, we cannot filter out any prefix of the Y_X input if the input sequence is $[x_5x_4x_3x_2x_1]$ during resumption. We now define some properties that guarantee that an input parameter Y_X receives the same sequence at resumption time.

We say that a transform Y is *in-det-out* if Y produces the same output sequence $\mathcal{Y}_O$ whenever it processes the same input sequences. All of Sagent's transforms are *in-det-out*.

Property 4.3 (**in-det-out(Y)**) Transform Y is *in-det-out* if Y produces the same output sequence whenever it processes the same input sequences. □

The in-det-out property guarantees that if a transform Y and all of the transforms preceding Y are in-det-out, and the data extractors produce the same sequences at resumption time, then Y will produce the same sequence, too. Hence, Z_Y receives the same sequence.

The requirement that all of the preceding transforms are in-det-out can be relaxed if some of the input parameters are *set-to-seq*. That is, if the order of the tuples in Y_X does not affect the order of the output tuples in Y_O, then Y_X is set-to-seq. For example, the same output sequence is produced by a sorting transform as long it processes the same set of input tuples.

Property 4.4 (**set-to-seq(Y_X)**) Given transform Y with input parameter Y_X, Y_X is *set-to-seq* if Y is in-det-out and $\forall \mathcal{Y}_X, \mathcal{Y}'_X$: ($\mathcal{Y}_X$ and $\mathcal{Y}'_X$ have the same set of tuples and all other input parameters of Y receive the same sequence) $\Rightarrow$ $Y(...\mathcal{Y}_X...) = Y(...\mathcal{Y}'_X...)$. □

Same-seq(Y_X) is a transitive property based on in-det-out and set-to-seq that holds if Y_X is guaranteed to receive the same sequence at resumption time. A weaker guarantee that sometimes allows for prefix filtering is that Y_X receives a suffix of the normal operation input $\mathcal{Y}_X$. We do not develop this weaker guarantee here.

Definition 4.4 (**Same-seq(Y_X)**) If X is an extractor then Same-seq(Y_X) = true if X uses the GetAllInorder re-extraction procedure. Otherwise, Same-seq(Y_X) = true if X is in-det-out and $\forall X_V$: Same-seq(X_V) or (X_V is set-to-seq and Same-set(V)). Otherwise, Same-seq(Y_X) = false. □

4.2 Identifying Contributors

To determine which $\mathcal{Y}_X$ tuples contribute to a warehouse tuple w_k, we are only provided with the value of w_k after the failure. Since transforms are black boxes, the only way to identify the contributors to w_k is to match the attributes that the $\mathcal{Y}_X$ tuples and w_k have in common. (If a transform changes an attribute value, e.g., reorders its bytes, we assume that it also changes the attribute name.)

We now define properties that, when satisfied, guarantee that we can identify exactly the $\mathcal{Y}_X$ contributors to w_k by matching certain *identifying attributes*, denoted IdAttrs(Y_X). In practice, some inconsequential $\mathcal{Y}_X$ input tuples may also match w_k on IdAttrs(Y_X). However, these tuples can be safely filtered since they do not contribute to the output. If the contributors cannot be identified, IdAttrs(Y_X) is set to [ ].

We define the *no-hidden-contributor* property to hold for Y_X if all of the $\mathcal{Y}_X$ tuples that contribute to some output tuple y_j match y_j on Attrs(Y_X) $\cap$ Attrs(Y_O). Selection, projection, aggregation, and union transforms have input parameters with no hidden contributors, as do all of Sagent's input parameters.

Property 4.5 (**no-hidden-contributor(Y_X)**) Given transform Y with input parameter Y_X, if $\forall \mathcal{Y}_X$, $\forall \mathcal{Y}_O$, $\forall y_j \in \mathcal{Y}_O$, $\forall x_i \in$ Contributors($\mathcal{Y}_X$, y_j), $\forall a \in$ (Attrs(Y_X) $\cap$ Attrs(Y_O)): ($Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$) $\Rightarrow$ ($x_i.a = y_j.a$) then *no-hidden-contributors*(Y_X) = true. □

If Y_X has no hidden contributors, we can identify a set of input tuples that contains all of the contributors to an output tuple y_j. This set is called the *potential contributors* of y_j. Shortly, we will use keys and other properties to verify that the set of potential contributors of y_j contains only tuples that do contribute to y_j. We now illustrate how the potential contributors are found.

Example 4.1 In the tree in Figure 4, the labels above the edges give the attributes of the input tuples, e.g., Attrs(Y_X) = [*cd*]. If Y_X has no hidden contributors, then all of the $\mathcal{Y}_X$ contributors to warehouse tuple w_k, denoted S_k, match w_k on [*cd*] (i.e., Attrs(Y_X) $\cap$ Attrs(Y_O)). If X_E has no hidden contributors, then the $\mathcal{X}_E$ contributors to $x_i \in S_k$ match x_i on [*c*] (i.e., Attrs(X_E) $\cap$ Attrs(X_O)). Therefore, the potential contributors of w_k in $\mathcal{X}_E$ are exactly the ones that match w_k on [*c*]. □

We call attributes that identify the $\mathcal{Y}_X$ potential contributors the *candidate identifying attributes* (CandAttrs) of Y_X.

Definition 4.5 (**CandAttrs(Y_X)**) There are three possibilities for CandAttrs(Y_X): (1) If Y is the warehouse inserter, CandAttrs(Y_X) = Attrs(Y_X). (2) If Y_X has hidden contributors, CandAttrs(Y_X) = [ ]. (3) Otherwise, CandAttrs(Y_X) = CandAttrs(Z_Y) $\cap$ Attrs(Y_X). □

In summary, CandAttrs(Y_X) is the set of attributes that are present throughout the path from Y_X to the warehouse, unless some input parameters has hidden contributors.

CandAttrs(Y_X) may identify both tuples that do and do not contribute to w_k. To isolate the actual contributors of w_k, we need to use key attributes and the *no-spurious-output* property. The no-spurious-output property holds for transform Y if each output tuple y_j has at least one contributor from each input parameter Y_X. While this property holds for many transforms, including all but one of Sagent's, union transforms do not satisfy it.

Property 4.6 (**no-spurious-output**(Y)) A transform Y produces *no spurious output* if $\forall$ input parameters Y_X, $\forall \mathcal{Y}_X$, $\forall \mathcal{Y}_O$, $\forall y_j \in \mathcal{Y}_O$: $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O \Rightarrow$ Contributors($\mathcal{Y}_X$, y_j) $\neq$ [ ]. □

We now illustrate how key attributes, candidate attributes, and the no-spurious-output property combine to determine the identifying attributes (IdAttrs).

Example 4.2 In Figure 4, CandAttrs(X_E) = $[c]$ if X_E, Y_X, and W_Y have no hidden contributors. There are three possibilities for IdAttrs(X_E): (1) IdAttrs(X_E) = KeyAttrs(X_E) if KeyAttrs(X_E) $\subseteq$ CandAttrs(X_E) and both X and Y satisfy the no-spurious-output property. (2) IdAttrs(X_E) = KeyAttrs(W_Y) if KeyAttrs(W_Y) $\subseteq$ CandAttrs(X_E). (3) IdAttrs(X_E) = IdAttrs(Y_X) if IdAttrs(Y_X) $\subseteq$ CandAttrs(X_E).

To illustrate (1), suppose KeyAttrs(X_E) is $[c]$. If $w_k.c = 1$, any $\mathcal{X}_E$ tuple that contributes to w_k must have $c = 1$ since CandAttrs(X_E) = $[c]$. Since neither X nor Y has spurious output tuples, there is at least one $\mathcal{X}_E$ tuple that contributes to w_k. c is the key for X_E, so the one $\mathcal{X}_E$ tuple with $c = 1$ must be the contributor.

To illustrate (2), suppose KeyAttrs(W_Y) = $[c]$. If $w_k.c = 1$, any $\mathcal{X}_E$ tuple that contributes to w_k must have $c = 1$ since CandAttrs(X_E) = $[c]$. Since c is the key of W_Y, all $\mathcal{X}_E$ tuples with $c = 1$ must contribute to either w_k or to no warehouse tuples.

To illustrate (3), suppose IdAttrs(Y_X) = $[c]$. Then given a warehouse tuple w_k with $w_k.c = 1$, we can identify the $\mathcal{Y}_X$ contributors to w_k, denoted S_k, by matching their c attribute with 1. Since X_E has no hidden contributors (because CandAttrs(X_E) $\neq$ [ ]), a $\mathcal{X}_E$ tuple with $c = 1$ must contribute to a tuple $x_j \in S_k$ or to no tuple in $\mathcal{Y}_X$. Hence, we can identify exactly the $\mathcal{X}_E$ contributors to w_k by matching their c attribute values.

In summary, the key attributes of X_E, Y_X (or any other input parameter in the path from X_E to W_Y), or W_Y can serve as IdAttrs(X_E). These key attributes must be a subset of CandAttrs(X_E) to ensure that matching can be performed between the warehouse tuples and the $\mathcal{X}_E$ tuples. □

The example above provides the intuition for our definition of IdAttrs.

Definition 4.6 (**IdAttrs**(Y_X)) Let P be the the path from Y_X to the warehouse. There are three possibilities for IdAttrs(Y_X). (1) If (KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X) and $\forall Z_V \in P$: Z_V has no spurious output tuples), then IdAttrs(Y_X) = KeyAttrs(Y_X). (2) Let $Z_V \in P$ but $Z_V \neq Y_X$. If IdAttrs(Z_V) $\neq$ [ ] and IdAttrs(Z_V) $\subseteq$ CandAttrs(Y_X), then IdAttrs(Y_X) = IdAttrs(Z_V). (3) Otherwise IdAttrs(Y_X) = [ ]. □

Case (1) in Definition 4.6 uses the key attributes of Y_X as IdAttrs(Y_X). Case (2) checks if the IdAttrs of each input parameter can be used as IdAttrs(Y_X). In [7], we discuss heuristics for choosing the input parameter in P whose identifying attribute is used for IdAttrs(Y_X).

4.3 The Trades Example Revisited

In Table 1, we provide SQL definitions for the transform functions in our main example. These definitions are not available to *DR*, and in general, cannot be written in SQL. Here, however, we use SQL to help illustrate the properties satisfied by the input parameters and transforms. Both transforms in Table 1 are in-det-out.

Transform	*Function computed by transform*
DT	`select *` `from` DT_{TRD} `where` $date \geq 12/1/98$ `and` $date \leq 12/31/98$
AV	`select` $AV_{PTE}.company, AV_{PTE}.pe,$ `avg(`$AV_{DT}.volume$`) as` $avgvol$ `from` AV_{PTE}, AV_{DT} `where` $AV_{PTE}.company = AV_{DT}.company$ `and` $AV_{PTE}.pe \leq 4$ `group by` $AV_{PTE}.company, AV_{PTE}.pe$

Table 1: Properties and functions of transforms.

The first four columns of Table 2 show the attributes, keys, and properties declared for each input parameter. We now explain why the properties hold. DT reads each tuple in DT_{TRD} and only outputs the tuple if it has a date in December 1998. Therefore, DT_{TRD} is suffix-safe, since DT outputs tuples in the input tuple order. It is map-to-one because each input tuple contributes to zero or one output tuple. It is not set-to-seq, since a different order of input tuples will produce a different order of output tuples.

Transform AV reads each tuple in AV_{PTE} and, if its pe attribute is $\leq$ 4, it finds all of the trade tuples for the same company in AV_{DT}, which are probably not in order by company. AV then groups the AV_{DT} tuples and computes the average trade volume for each company. Then it processes the next tuple in AV_{PTE}. AV_{PTE} is map-to-one since each tuple contributes to zero or one output tuple. the same reason it is suffix-safe: AV processes tuples from AV_{PTE} in order. AV_{DT} is map-to-one since each trade tuple contributes to the average volume tuple of only one company. However, AV_{DT} is not suffix-safe, e.g., the trade tuple needed to join with the first tuple in AV_{PTE} may be the last tuple in AV_{DT}. Similarly, it is set-to-seq because the order of trades tuples is not relevant to AV. Finally, since the warehouse inserter stores its input tuples in order, W_{AV} is map-to-one and suffix-safe but not set-to-seq.

The last two columns of Table 2 show the identifying attributes and the transitive properties. None of the input parameters has hidden contributors. The identifying attribute of W_{AV}, AV_{DT}, DT_{TRD} and AV_{PTE} is $[company]$ because it is the key of W_{AV}. The transitive properties (e.g., Subset-feasible) are computed (by *DR*) using Definitions 4.2 and 4.3. Note that Same-seq and Same-set are not computed since the re-extraction procedures have not been determined yet.

5 The *DR* Resumption Algorithm

We now present the *DR* resumption algorithm, which uses the properties developed in Section 4. *DR* is actually two algorithms, *Design* and *Resume*, hence the name. After a component tree G is designed, *Design* constructs a

Input Y_X	*Attrs(Y_X)*	*KeyAttrs(Y_X)*	*Y_X Properties*	*IdAttrs(Y_X)*	*Y_X Transitive Properties*
DT_{TRD}	[date,company,volume]	[date,company]	map-to-one suffix-safe	[company]	Subset-feasible
AV_{PTE}	[company,pe]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible
AV_{DT}	[date,company,volume]	[date,company]	map-to-one set-to-seq	[company]	Subset-feasible
W_{AV}	[company,pe,avgvol]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible

Table 2: Declared and inferred properties of input parameters.

component tree G' that *Resume* uses to resume any failed warehouse load on G. The component tree G' is the same as G except: (1) re-extraction procedures are assigned to the extractors in G'; and (2) filters are assigned to some of the input parameters in G'.

Design constructs G' using only the declared attributes, keys, and properties of G. When a warehouse load that uses G fails, *Resume* initializes the filters and re-extraction procedures in G' based on the tuples that were stored in the warehouse. *Resume* then uses G' to resume the warehouse load. Since neither *Design* nor *Resume* runs during normal operation, *DR* does not incur any normal operation overhead!

5.1 Example using *DR*

We first illustrate *DR* on our running example. *Design* first computes the Subset-feasible and Prefix-feasible transitive properties and the IdAttrs of each input parameter, as shown in Figure 7.

Design then constructs G'. First, it assigns re-extraction procedures to the extractors based on their computed properties and identifying attributes. Since IdAttrs(AV_{PTE}) = $[company]$, $company$ can identify contributor source PE tuples. Since both Prefix-feasible(AV_{PTE}) and Subset-feasible(AV_{PTE}) hold, *Design* can assign either GetSuffix or GetSubset to PTE to avoid re-extracting all the PE tuples. Suppose PTE supports neither GetSuffix nor GetSubset. *Design* assigned GetAllInorder to PTE instead.

Design can assign GetSubset to TRD since Subset-feasible(DT_{TRD}) holds and IdAttrs(DT_{TRD}) = $[company]$. However, suppose TRD only supports GetAll, so it is assigned instead.

For each input parameter, *Design* then chooses whether to discard a prefix of the input ("prefix filter"), or to discard a subset of the input ("subset filter"). Since discarding a prefix requires the Same-seq property, *Design* computes the Same-seq property as it assigns filters to each input parameter, as follows.

DT_{TRD}: Same-seq(DT_{TRD}) does not hold because TRD is assigned GetAll, so it is not possible to filter a prefix of the DT_{TRD} input sequence. Since DT_{TRD} is Subset-feasible and IdAttrs(DT_{TRD}) = $[company]$, it is possible to assign a subset filter, denoted DT^f_{TRD}, to remove a subset of the DT_{TRD} input sequence. When a failed load is resumed, DT^f_{TRD} removes the subset of tuples in DT_{TRD} whose $company$ attribute value matches some warehouse tuple.

AV_{PTE}: Same-seq(AV_{PTE}) holds because PTE is assigned GetAllInorder. AV_{PTE} is also Prefix-feasible. Therefore, a prefix filter AV^f_{PTE} is assigned to AV_{PTE}. When a failed load is resumed, AV^f_{PTE} removes the prefix of the AV_{PTE} input sequence that ends with the tuple whose $company$ attribute matches the last warehouse tuple.

AV_{DT}: AV_{DT} is Subset-feasible and IdAttrs(AV_{DT}) = $[company]$, so a subset filter can also be assigned to AV_{DT}. It is not assigned, however, since *Design* determines that this filter is redundant with the DT^f_{TRD} filter.

Figure 8 shows G'. G' is constructed in two "passes" over G: a backward pass to compute IdAttrs, Prefix-feasible, and Subset-feasible, and a forward pass to compute Same-seq and assign filters. Hence, the time to construct G' is negligible compared to the time to design and debug G, which is on the order of weeks or months [10]. Algorithm *Design* is done and G' is set aside.

Now suppose that a load using G fails, and the tuple sequence that made it into the warehouse is $\mathcal{C} = [\ \langle AAA, 3, 12500\rangle, \langle INTC, 2, 98000\rangle, \langle MSN, 4, 15000\rangle\ ]$, where the three attributes are $company$, pe, and $avgvol$, respectively. Based on $\mathcal{C}$, *Resume* instantiates the filters and procedures (i.e., GetSuffix, GetSubset) of G' that are sensitive to $\mathcal{C}$.

Subset filter DT^f_{TRD} is instantiated to remove all DT_{TRD} tuples whose company is AAA, $INTC$ or MSN. Prefix filter AV^f_{PTE} is instantiated to remove the prefix of its AV_{PTE} input that ends with the tuple whose $company$ attribute is MSN. Then the load is resumed by calling the re-extraction procedures of G'. Because of the filters, the input tuples that contribute to the tuples in $\mathcal{C}$ are filtered and are not processed again by DT, AV and W.

In summary, *DR* avoids re-processing many of the input tuples using filters. Also, if the extractors PTE and TRD had supported GetSubset or GetSuffix, *DR* could have even avoided re-extracting tuples from the sources.

5.2 Filters

In the example, we mentioned subset filters and prefix filters. There are two types of subset filters and two types of prefix filters that may be assigned to Y_X. In each case, the filter

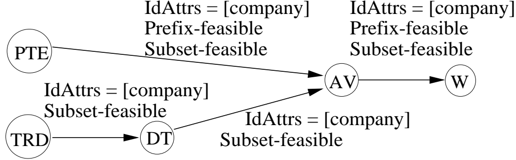

Figure 7: IdAttrs and transitive properties of G

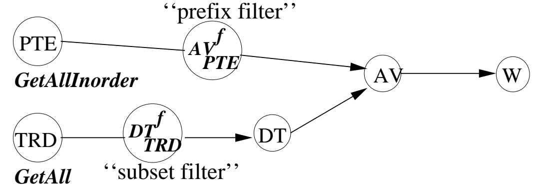

Figure 8: G' with procedures and filters assigned

receives X's output sequence as input, and the filter sends its output to Y as the Y_X input sequence.

Clean-Prefix Filter: The clean-prefix filter $CP[s, A]$ is instantiated with a tuple s and a set of attributes A. CP discards tuples from its input sequence until it finds a tuple t that matches s on A. CP discards t, and continues discarding until an input tuple t' does *not* match s on A. All tuples starting with t' are output by CP. We use CP on Y_X when Y_X is Subset-feasible, Prefix-feasible, and Same-seq, and IdAttrs(Y_X) is not empty. In this case, all input tuples up to and including the contributors of the last $\mathcal{C}$ tuple, denoted Last($\mathcal{C}$), can be safely filtered. So CP is instantiated as $CP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$, where $\mathcal{C}$ is the tuple sequence in the warehouse after the crash. We call CP a clean filter because no $\mathcal{C}$ contributors emerge from it.

Dirty-Prefix Filter: The dirty-prefix filter $DP[s, A]$ is a slight modification to CP that begins its output sequence with t instead of t'. We use DP on Y_X when Y_X is Prefix-feasible and Same-seq, but not Subset-feasible. DP is instantiated as $DP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$.

Clean-Subset Filter: The clean-subset filter $CS[\mathcal{S}, A]$, is instantiated with a tuple sequence $\mathcal{S}$ and a set of attributes A. For each tuple t in its input sequence $\mathcal{I}$, if t matches any $\mathcal{S}$ tuple on the A attributes, then t is discarded. Otherwise, t is output. In other words, CS performs an anti-semijoin between $\mathcal{I}$ and $\mathcal{S}$ ($\mathcal{I}\overline{\ltimes}_A\mathcal{S}$). We use CS on Y_X when Y_X is Subset-feasible and IdAttrs(Y_X) is not empty. CS is instantiated as $CS[\mathcal{C}, \text{IdAttrs}(Y_X)]$.

Dirty-Subset Filter: The dirty-subset filter $DS[\mathcal{C}, A]$, is a slight modification to CS that applies when Y_X is Prefix-feasible but not Same-seq. Unlike CS, DS removes a suffix $\mathcal{C}_s$ of $\mathcal{C}$ before performing the anti-semijoin. $\mathcal{C}_s$ contains the tuples that share Y_X contributors with Last($\mathcal{C}$) and can be obtained by matching $\mathcal{C}$ tuples with the Last($\mathcal{C}$) tuple on IdAttrs(Y_X). DS then acts like the clean-subset filter $CS[\mathcal{C} - \mathcal{C}_s, \text{IdAttrs}(Y_X)]$.

In summary, the properties that hold for an input parameter Y_X determine the types of filters that can be assigned to Y_X. When more than one filter type can be assigned, we assign the filter that removes the most input tuples. When filter type f removes more tuples than g, we say $f \succ g$. The relationships among the filter types are as follows: $CP \succ DP \succ DS,\ CP \succ CS \succ DS$.

Hence, we try to assign the clean-prefix filter first, and the dirty-subset filter last. In *DR*, we assign the dirty-prefix filter before the clean-subset filter for two reasons. First, it is much cheaper to match each input tuple to a single filter tuple s than to a sequence of tuple filters $\mathcal{S}$. Second, the prefix filters can remove tuples that do not contribute to any warehouse tuple, simply because they precede a contributing tuple. The subset filters can only remove contributors. The second advantage is especially apparent in our experimental results in Section 6.

Algorithm 5.1 *AssignFilter*
Input: Component trees G, G', input parameter Y_X
Output: Input parameter Y_X in G' is assigned a filter whenever possible
  If (Prefix-feasible(Y_X) and Subset-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ])
    Insert $Y_X^f = CP$[Last($\mathcal{C}$),IdAttrs(Y_X)] between Y and X (in G')
  Else If (Prefix-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ])
    Insert $Y_X^f = DP$[Last($\mathcal{C}$), IdAttrs(Y_X)] between Y and X
  Else if (Subset-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ])
    Insert $Y_X^f = CS[\mathcal{C}$, IdAttrs(Y_X)] between Y and X
  Else if (Prefix-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ])
    Insert $Y_X^f = DS[\mathcal{C}$, IdAttrs(Y_X)] between Y and X

Figure 9: AssignFilter algorithm

The procedure *AssignFilter* is shown in Figure 9. Observe that *AssignFilter* assigns a filter to Y_X whenever possible. *Design* uses a subsequent procedure to remove redundant filters.

5.3 Re-extraction Procedures

The re-extraction procedures are now defined in terms of the filters.

Definition 5.1 (**Re-extraction procedures**)
GetAllInorder() $= \mathcal{E}_O$, the output of E during normal operation.
GetAll() $= \mathcal{T}$: $\mathcal{T}$ and $\mathcal{E}_O$ have the same set of tuples.
GetSuffix(s,A) $= \mathcal{T}$: $CP[s,A] = \mathcal{T}$.
GetDirtySuffix(s,A) $= \mathcal{T}$: $DP[s, A] = \mathcal{T}$.
GetSubset($\mathcal{S}$,A) $= \mathcal{T}$: $CS[\mathcal{S}, A] = \mathcal{T}$.
GetDirtySubset($\mathcal{S}$,A) $= \mathcal{T}$: $DS[\mathcal{S}, A] = \mathcal{T}$. □

The procedure *AssignReextraction*, which is shown in [7], is similar to *AssignFilter*. It tries to push the filters into the re-extraction. For example, the same properties that allow CP to be assigned to E_O also allow GetSuffix to be assigned to E instead.

Algorithm 5.2 *Design*
Input: Component tree G
Output: Component tree G'
1. Copy G to G'
2. Compute IdAttrs(Y_X), Subset-feasible(Y_X), Prefix-feasible(Y_X) for each input parameter Y_X in reverse topological order.
3. For each extractor E
   AssignReextraction(G,G',E)
4. For each input parameter Y_X in topological order
   Compute Same-seq(Y_X)
   AssignFilter(G,G',Y_X)
   If Y_X is assigned a filter,
      Set Same-seq(Y_X) to false.
5. *RemoveRedundantFilters*(G, G')
6. Save G' persistently and return G'

Algorithm 5.3 *Resume*
Input: Component tree G'
Side Effect: Resumes failed warehouse load
Let $\mathcal{C}$ be the tuples in the warehouse
1. Instantiate each re-extraction procedure in G' and each filter in G' with actual value of $\mathcal{C}$
2. For each extractor E in G'
   Invoke re-extraction procedure assigned to E

Figure 10: *DR* algorithm

5.4 The *Design* and *Resume* Algorithms

Figure 10 shows algorithms *Design* and *Resume* of *DR*. *Design* constructs G' by processing the component tree G in two passes. In reverse topological order, *Design* computes IdAttrs, Prefix-feasible and Subset-feasible. It then uses *AssignReextraction* to assign procedures to the extractors. Finally, in topological order, it computes Same-seq and uses *AssignFilters* to assign filters to the input parameters. In [7], we show how *Design* removes redundant filters. After a failure, *Resume* simply instantiates the re-extraction procedures and filters in G' with the actual value of the warehouse tuple sequence $\mathcal{C}$. The warehouse load is then resumed by invoking the re-extraction procedures.

The worst-case complexity of *DR* is $O(n^2 \cdot |\mathcal{C}| + n^3)$, assuming that n^2 filters are assigned. However, in practice, few filters are assigned by *DR*, but those filters lead to significant performance improvements. Furthermore, our experiments show that the overhead in instantiating the filters is reasonable.

6 Experiments

In this section, we compare *DR* to other recovery algorithms. We performed experiments using Sagent's Data Mart 3.0. The software ran on a Dell XPS D300 with a Pentium II 300 MHz processor and 64 MB of RAM. More details of the experiments are in [7].

We first examined the extractors and transforms offered by Sagent. All of the extractors support GetAll and GetAllInorder, but only the "SQL" extractor support GetSuffix, GetDirtySuffix, GetSubset, and GetDirtySubset. The properties of the 19 basic Sagent transforms (selection, projection, union, aggregation, join, etc.) are summarized in Figure 11, and show that the properties we have defined are fairly common in practice.

Transforms:
100% (19/19) are in-det-out.
95% (18/19) have no spurious output.
Input Parameters:
91% (21/23) are map-to-one.
78% (18/23) are suffix-safe.
17% (4/23) are set-to-seq (i.e., perform sorting).
100% (23/23) have no hidden contributors.

Figure 11: Properties of Sagent transforms and input parameters

We then constructed two different component trees. The first tree loads the TPC-D fact table *Lineitem*. Fact tables typically extract transactional data from a single table, massage it, in this case with four transforms, and store it in the warehouse. The second tree loads the materialized view corresponding to TPC-D query Q3, which uses five transforms to join three source tables and perform a GROUP BY and a SUM of revenue estimates.

6.1 *DR* versus no-overhead algorithms

In the first set of experiments, we compared *DR* to the algorithms that impose no normal operation overhead (i.e., the lower right quadrant of Figure 2). We compared three variants of *DR* to *Redo* and *Inf*. *Inf* is the algorithm used by Informatica [6], and uses only one filter just before the inserter. DR_{src} pushes filtering to the extractors. DR_{pre} uses a prefix filter right after each extractor, while DR_{sub} uses a subset filters there. Normally, *DR* would produce DR_{src}.

Figure 12 plots resumption time for the fact table load against the number of tuples already loaded in the warehouse, which we varied from 0–95%. As expected, *DR* performs better than *Redo* when 20% (or more) of the tuples are in the warehouse. For instance, when the warehouse is 95% loaded, DR_{src} completes a resumed load 10 times faster than *Redo*, and and DR_{sub} is 2.3 times faster. *DR* is also faster than *Inf*. On the other hand, when the warehouse is empty at the time of failure, *DR* performs 10–12% worse than *Redo* because of the filter overhead. Among the three *DR* variants, DR_{src} performs the best because it filters the tuples the earliest. DR_{sub} performs worse than DR_{pre} because of the expensive anti-semijoin operation employed by DR_{sub}'s filters.

Figure 13 shows similar results for the view load. However, DR_{sub} and *Inf* perform worse than *Redo* regardless of how many tuples are loaded: the view query is very selective, and many of the source tuples extracted do not contribute to any warehouse tuple. Since subset filters can only remove tuples that contribute to a warehouse tuple, the filters used by

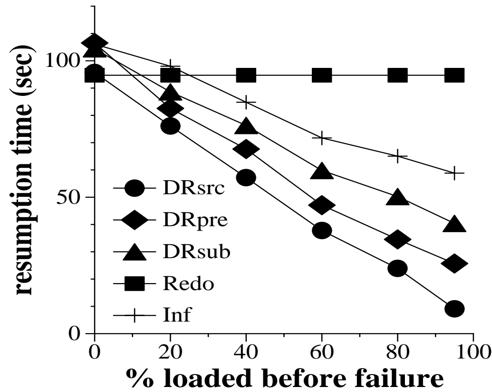

Figure 12: Fact table load resumption

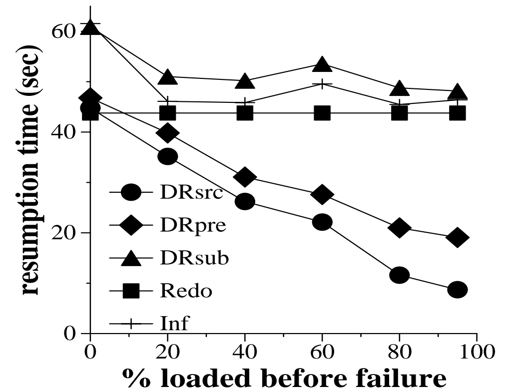

Figure 13: View table load resumption

# Savepoints	Load Time (s)	% Increase
0	94.7	0%
1	166.4	75.7%
2	245.9	159.7%
3	314.0	231.6%

Table 3: Fact table savepoint overhead

# Batches	Load Time (s)	% Increase
1	94.7	0%
2	97.6	3.1%
3	104.8	7.4%
4	107.0	13.0%
5	113.0	19.3%
10	150.6	59.0%

Table 4: Fact table batching overhead

DR_{sub} and *Inf* do not remove enough tuples to compensate for the cost of their filters.

6.2 *DR* versus Savepoints and Batching

In the second set of experiments, we compared *DR* to "staging" (with savepoints) and "batching," two algorithms in the upper left quadrant of Figure 2.

Table 3 compares the normal operation overhead of loading the fact table as savepoints are added. Without savepoints, the fact table loads in 94.7 seconds. Each savepoint slows down the load by 70-80%! In contrast, *DR* has no overhead.

We also introduced up to three savepoints to the view load. However, even with three savepoints, the load is only 7% slower because the "*Join*" transforms are very selective and occur before the first savepoint. Hence, few tuples are recorded in the savepoints.

Table 4 shows the normal operation overhead of batching. As the load is divided into more batches, pipelining decreases and there is more overhead for starting batches, so the total load time increases.

We then measured the resumption times of *DR*, using DR_{src}, versus *Save*, which used two savepoints, and *Batch*, which used three batches. We loaded the warehouse and stopped the load after t_{fail} seconds. We then resumed the load and recorded the resumption time.

Figures 14 and 15 plot the resumption times of *DR*, *Save*, and *Batch* as t_{fail} increases. The graphs show that *Save*'s resumption time improves in discrete steps that occur as each savepoint completes. For the fact table load, *DR* is more efficient than *Save* because the warehouse table is populated early in the load, and *DR* can use the warehouse tuples to make resumption efficient. *DR*'s resumption time for the view load is relatively slower because the first output tuples are not produced until the load is nearly complete. Unfortunately, neither *Save* nor *DR* performs well. The two savepoints *Save* uses essentially partition the view tree into three "sub-trees." Although *Save* does not use incomplete savepoints to improve resumption, *DR* can treat an incomplete savepoint and the "sub-tree" that produced it as a warehouse table and a component tree. The performance of this hybrid algorithm, denoted *Hybrid*, is better than either *Save* or *DR*.

Batch's resumption time also improves in discrete steps, as each batch completes. *DR* is surprisingly more efficient than *Batch* on the fact table load, given that *DR* imposes no normal operation overhead. However, *DR* can use any completed subset of the load to reduce resumption time. On the view load, *Batch* resumes faster than *DR*: *Batch*'s resumption speed is its payback for "manual" modification of the workflow and much slower normal operation loads.

6.3 Discussion

We can draw a number of conclusions from the previous experiments. First, *DR* resumes a failed load much more efficiently than *Redo* and *Inf*. *DR* is also flexible in that the more properties exist, the more choices *DR* has and the better *DR* performs.

Second, there is a need for a "cost-based" analysis of when to use *DR*. For instance, if the warehouse table is empty, *Redo* is better than both *DR* and *Inf*. However, as more tuples are loaded, using *DR* becomes more beneficial. A "cost-based" analysis can also determine when to use subset

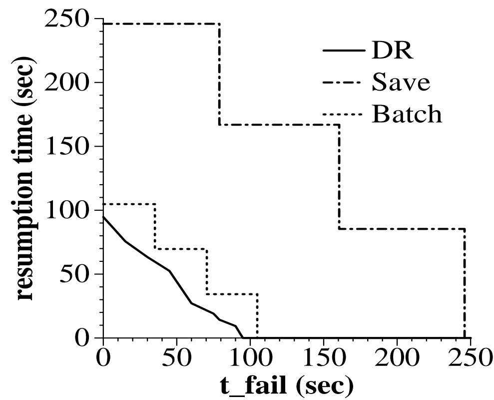

Figure 14: Savepoints and Batching vs DR (Fact table)

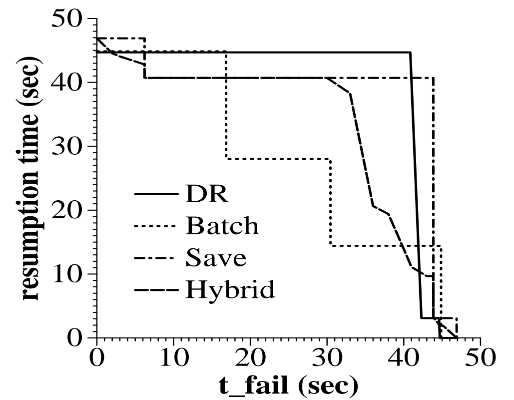

Figure 15: Savepoints and Batching vs DR (View table)

filters, which may not remove enough tuples to justify their cost (e.g., the cost of performing an anti-semijoin).

Third, savepoints (or snapshots) result in significant normal operation overhead. However, if certain transforms of a component tree are very selective (i.e., few output tuples compared to input tuples), the overhead of savepoints may be tolerable. When a batching algorithm is used, a careful selection of the number of input batches is required: More batches can result in significant normal operation overhead. On the other hand, fewer batches result in longer resumption times.

7 Conclusions

We developed a resumption algorithm *DR* that performs most of its analysis during "design time," and imposes no overhead during normal operation. The *Design* portion of *DR* only needs to be invoked once, when the warehouse load component tree is designed, no matter how many times the *Resume* portion is called to resume a load failure. *DR* is novel because it uses only properties that describe how complex transforms process their input at a high level (e.g., are the tuples processed in order?). These properties can be deduced easily from the transform specifications, and some of them (e.g., keys, ordering) are already declared in current warehouse load packages. By performing experiments under various TPC-D scenarios using Sagent's load facility, we showed that *DR* leads to very efficient resumption.

Although we have developed *DR* to resume warehouse loads, *DR* is useful for many applications. In particular, if an application performs complex and distributed processing, *DR* is a prime recovery algorithm candidate when minimal normal operation overhead is required. Since previous algorithms either require heavy overhead during normal operation, or incur high recovery cost, *DR* fills the need for an efficient lightweight recovery algorithm.

References

[1] P. A. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *SIGMOD*, pp. 112–122, 1990.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kaufman, 1997.

[3] F. Carino. High-performance, parallel warehouse servers and large-scale applications, Oct. 1997. Talk about Teradata given in Stanford Database Seminar.

[4] TPC Committee. Transaction Processing Council. Available at: http://www.tpc.org/.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufman, 1993.

[6] Informatica. Powermart 4.0 overview. Available at: http://www.informatica.com/pm_tech_over.html.

[7] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Resumption algorithms. Technical report, Stanford University, 1998. Available at http://www-db.stanford.edu/pub/papers/resume.ps.

[8] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *SIGMOD*, pp. 361–370, 1992.

[9] R. Reinsch and M. Zimowski. Method for Restarting a Long-Running, Fault-Tolerant Operation in a Transaction-Oriented Data Base System Without Burdening the System Log. U.S. Patent 4,868,744, IBM, 1989.

[10] Sagent Technologies. Personal correspondence with customers.

[11] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *VLDB*, pp. 30–41, Zurich, Switzerland, 1995.

[12] A. Witkowski, F. Cariño, and P. Kostamaa. NCR 3700 — The Next-Generation Industrial Database Computer. In *VLDB*, pp. 230–243, 1993.