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 simple transformations. We develop a resumption algorithm called *DR* that imposes no overhead and relies only on the basic properties of the transformations. We show that *DR* can lead to almost a ten-fold reduction in resumption time by performing experiments using commercial software to load TPC-D tables and materialized views.

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 can range from 1 to 100 Gb and take up to 24 hours to execute. For example, Walmart's maintenance load averages 16 Gb per day [Car97], and typical Sagent customer maintenance loads process 6 Gb weekly, and up to 100 Gb to create the warehouse.

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 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 If there is not enough time for the new load, it may be skipped, leaving the database out of date or incomplete, and generating an even bigger load for the next period.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 every thirty loads fails [Inc].

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, very low-overhead, technique for *resuming* failed loads. Our technique exploits some generic "properties" of the workflow used to load the warehouse, so that work is not repeated during a resumed load.

*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.

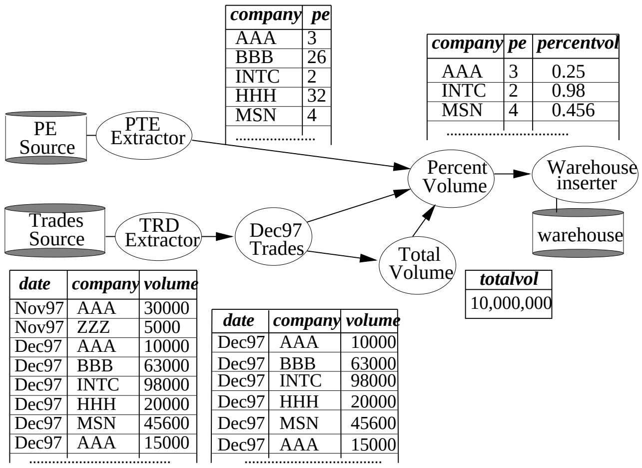

Figure 1: Load Workflow

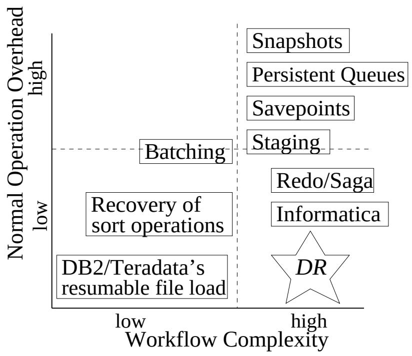

Figure 2: Applicability of Algorithms

To illustrate the type of processing performed during a load, consider the 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 prefix of the tuples extracted from each source. The stock trade data is first processed by the *Dec97Trades transform*, which only outputs trades from December 1997. Thus, the first two trades are removed since they happened in November 1997. The *TotalVolume* transform then computes the total volume of the December 1997 trades. The *PercentVolume* transform then groups the trades by company and finds the percent of the total trade volume contributed by companies whose *pe* is less than or equal to 4. For instance, companies BBB and HHH are discarded since they have high *pe*'s. An AAA tuple is output since its *pe* value is low: its *percentvol* value is the sum of the AAA volumes (25,000, assuming all AAA tuples are shown in the figure) divided by the *TotalVolume* output. The output of *PercentVolume* 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 [Inc]. The transforms can do arbitrary processing (e.g., data scrubbing, byte reordering) and are often written by application specialists. In most scenarios it is not feasible to ask these specialists to add recovery code to their transforms. Similarly, it is not desirable to add high overhead state-saving transforms to aid recovery.

Therefore, it is difficult to use traditional techniques that deal with interrupted long-duration tasks. One common technique is to break the input into batches, and to process each batch in sequence. If the load fails during the fourth batch, for example, then only the fourth batch needs to be repeated (not the first three), and then the load can continue. However, it is very difficult to identify batches that can be processed independently when dealing with an arbitrary load workflow. Even if the semantics of the load workflow are known, in many cases, the input cannot be split. In our example in Figure 1 *TotalVolume* requires the entire input before it generates its output: batches cannot be formed.

Another technique is to take periodic snapshots or savepoints [GR93] of the workflow state. When a failure occurs, each transform reverts to the latest savepoint, and proceeds from there. This technique requires that the semantics of the transforms be known, and that the transforms be modified to take and restore savepoints. To implement savepoints, we need to save tuples in transit between transforms in persistent queues [BHM90,BN97]. The use of savepoints is often not feasible because transforms cannot be modified, and because of the overhead of taking savepoints and using persistent queues.

A third technique is to divide the workflow into consecutive stages, and save intermediate results. All input data enters the first stage. All of the first stage's output is saved. The saved output then serves as input to the second stage, and so on. If a failure occurs while the second stage is active, it can be restarted, without having to redo the work performed by the first stage. Staging is not desirable due to the overhead of saving intermediate results and because of the loss of parallelism (tuples are not pipelined between stages).

With the technique we propose in this paper, there is no overhead during normal operation, and transforms do not need to be modified. The transform writer must declare whether the transform satisfies some simple properties (e.g., are tuples processed in order?) that are used at recovery time. The properties are fairly simple and 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. If we know that tuples are processed in alphabetical order by the *PTE Extractor* and by the *PercentVolume* 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. Our scheme is not always able to eliminate tuples during reprocessing; however, it does offer significant improvements in many cases, as in this example. 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 essence, our strategy is to exploit some basic semantics of the load workflow, and to be selective when resuming a failed load. The fundamental features of our technique are informally contrasted to others in Figure 2. The vertical axis represents the overhead of a technique during normal operation, while the horizontal axis indicates whether the technique can deal with general workflows. The lower left quadrant contains resumption schemes for very specific applications like disk-based sorting [MN92] and object database loading [WN95] because they are efficient during normal processing. The resumable bulk load utilities provided by DB2 and Teradata [RZ89,WCK93] are in the same category because they are only able to resume a load of a flat file copied into the warehouse.

The batching technique discussed earlier is near the top left quadrant because our experiments in Section 6 show that it can be inefficient during normal processing, yet it is only applicable to workflows whose input can be broken into independent batches.

In the top right quadrant are techniques that are general but incur high overhead. We believe that most traditional fault-tolerant techniques, including savepoints and persistent queues [GR93] fall in this category. In addition, they often require that transforms be modified so that they cooperate during recovery (e.g., by restoring a savepoint).

In the lower right quadrant are low-overhead, general solutions. Redoing the entire load after a failure is in this category. It has low overhead during normal processing, but incurs a very high cost during recovery. Informatica's solution [Inf] is similar: After a failure, Informatica reprocesses the data in its entirety, only filtering out the already stored tuples when they reach the warehouse for the second time (i.e., just before the inserter). Sagas [GMS87,GR93] also incur high cost during recovery because the load must be restarted from the beginning. Our algorithm, called *DR*, is also in this quadrant but has a much lower recovery cost. *DR* intercepts and removes the redundant tuples as early as possible, achieving greater efficiencies at recovery time.

In summary, in this paper 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 an efficient resumption algorithm (*DR*) that filters input tuples as early as possible, sometimes not even re-extracting the tuples from the sources. *DR* imposes no overhead during normal operation, and requires no changes to the transforms. Also, *DR* does not require knowing the specifics of a transform, but only its basic properties, such as key attributes. *DR* is presented here in the context of warehousing, but is really a generic solution for resuming any long-duration, data intensive task.
- We show experimentally that *DR* can significantly reduce load times when a failure occurs, 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.

The rest of the paper is organized as follows. We describe a normal warehouse load in Section 2, and discuss warehouse load failure in Section 3. We develop the *DR* resumption algorithm in Sections 4 and 5. Experiments are presented in Section 6 and we conclude in Section 7.

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 $[a_1..a_m]$. □

Before we describe a successful warehouse load, we discuss how a component directed acyclic graph (DAG) represents a load workflow, and how it is designed.

2.1 Component DAG Design

Figure 3 illustrates the same component DAG as Figure 1, with abbreviations for the transform names. Constructing a component DAG 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(s) into which the inserter loads the data are specified. The extractors, transforms, and inserter comprise the nodes of the DAG.

Each transform and inserter expects certain *input parameter* sequences at load time. The components that supply these input parameters are also specified when the component DAG is designed. Similarly, each transform and extractor generates an output sequence to its *output parameter*. In commercial packages, the input and output parameters are specified by connecting the extractors, transforms, and the inserter together with edges in the component DAG.

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

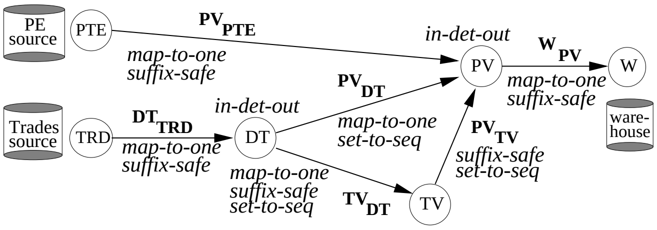

Figure 3: Component DAG with Properties

As a component DAG is designed, the "properties" that hold for the various transforms and their input parameters are declared for use by our resumption algorithm. Commercial load packages already declare basic properties like tuple sequence keys. The properties that DR uses are explained in more detail in Section 4. We now illustrate a component DAG.

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 $Dec97Trades$), TV (for $TotalVolume$), and PV (for $PercentVolume$). The inserter is denoted W.

The input parameter(s) of each component are denoted by the component that produces the input. For instance, PV_{DT} is an input parameter of PV that is produced by DT. Each extractor and transform also has an output parameter although they are not shown in Figure 3. For instance, the output parameter of DT is denoted DT_O. In Figure 3, DT_O is used as input by PV and TV. That is, $PV_{DT} = DT_O$ and $TV_{DT} = DT_O$.

Figure 3 also shows the properties that hold for each input parameter and each transform. For instance, the properties *map-to-one*, *suffix-safe* and *set-to-seq* hold for input parameter TV_{DT}, and the property *in-det-out* holds for transform TV. In Section 4, we define these properties and justify why they hold in this example. We then use them in DR in Section 5. When the component DAG is designed, the attributes and keys of input parameters are also declared. For instance, the attributes of PV_{PTE} tuples are $[company, pe]$, while the keys are $[company]$. □

In summary, 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 keys. W denotes the warehouse inserter.

We note that the component DAGs designed for warehouse creation and maintenance are different. Component DAGs for creation perform the initial population of tables, while component DAGs for maintenance typically populate "delta" tables with changes that are later applied to the tables. (Creating separate delta tables allows standard view maintenance algorithms to be applied to materialized views over these tables.) While a component DAG for creation only handles new tuples, a maintenance component DAG handles tuple inserts, deletes, and updates. Therefore, the components used are usually different. However, our resumption algorithm applies equally well to both creation and maintenance component DAGs. Thus, in the rest of the paper, when we refer to a "load," it could be for initial warehouse creation or for warehouse maintenance.

2.2 Successful Warehouse Load

When a component DAG is used to load data, the extractors produce sequences that serve as 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, inserts the tuples in batches, and periodically issues a commit command to ensure that the tuples are stored persistently. Note that each component's output sequence can be received as the next component's input as it is generated, to maximize pipelined parallelism. 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 components. The next example illustrates a warehouse load during normal operation, i.e., no failures occur.

Example 2.2 Consider the component DAG in Figure 3. First, extractors fill their output parameters PTE_O and TRD_O with the sequences $\mathcal{PTE}_O$ and $\mathcal{TRD}_O$, respectively. (The calligraphy font denotes sequences.) Input parameter PV_{PTE} is instantiated with the sequence $\mathcal{PV}_{PTE} = \mathcal{PTE}_O$. Similarly, DT_{TRD} is instantiated with $\mathcal{DT}_{TRD} = \mathcal{TRD}_O$, and so on. Finally, W_{PV} of the inserter is instantiated with $\mathcal{W}_{PV} = \mathcal{PV}_O$. W inserts the tuples in $\mathcal{W}_{PV}$ in order and issues a commit periodically. In the absence of failures, $\mathcal{W}_{PV}$ 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 $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$. We also use $\mathcal{W}$ to denote the sequence that is loaded into the warehouse in the absence of failures.

3 Warehouse Load Failure

In general, there are two types of failures that can prevent a load from completing - logical failures (e.g., invalid data) and system-level failures (e.g., RDBMS or software crashes, hardware crashes, lack of disk space). If a load fails because of invalid data, the load will again fail if it is restarted to process the same invalid data. On the other hand, if a load fails because of system-level failures, it is not likely that the load will fail once it is restarted. This assumes of course that the necessary actions were taken to fix the failure, e.g., software was restarted, or the hardware was fixed/replaced, or disk space was allocated. In this paper, we focus on system-level failures. Furthermore, we consider system-level failures that do not affect information stored in stable storage.

3.1 Component Failures

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. □

We now show why this observation holds for each type of component failure. When a source or its extractor E fails, only a prefix of E's normal operation output has been produced. Let transform Y take the output of E as its input. Y therefore receives and processes only part of its normal input and produces only a prefix of its output. Any transform Z that receives Y's output will then produce a prefix of its output, etc. This cascade of incomplete inputs eventually reaches the warehouse inserter W, causing it to insert only a prefix of $\mathcal{W}$.

Similarly, when a transform Y fails, only a prefix of Y's output has been produced. Again, a cascade of incomplete inputs leads to a prefix of $\mathcal{W}$ being stored in the warehouse. Finally, when the warehouse or the inserter W fails, it is clear that only a prefix of $\mathcal{W}$ is inserted and committed by W into the warehouse. (Note that the prefix may be empty.)

A network failure between components Y and Z results in only a prefix of Y's output reaching Z. Therefore, the effect of a network failure is the same as if component Y had failed. Henceforth, we ignore network failures since they can be modeled as failures of other components.

3.2 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. However, only limited data is available to the resumption algorithm, because the transforms are stateless and store no data. The resumption algorithm may use the prefix of the warehouse input $\mathcal{W}$ that is in the warehouse. In addition, the following procedures (and other slight variants) may be provided by each extractor E. We use $\mathcal{E}_O$ to denote the sequence that would have been extracted by E had there been no failures. More details on all of the re-extraction procedures are provided in Section 5.3.

- GetAllReordered() extracts the same set of tuples as the set of tuples in $\mathcal{E}_O$. The order of the tuples may be different because many sources, such as commercial RDBMS, do not guarantee the order of the tuples. We assume that all extractors provide GetAllReordered(), that is, that the original data is still available. If $\mathcal{E}_O$ cannot be reproduced, then $\mathcal{E}_O$ must be logged.
- GetAll() extracts the same sequence $\mathcal{E}_O$. This procedure may be supported by an extractor of a commercial RDBMS that initially extracted tuples with an SQL ORDER BY clause. Thus, the same tuple order can be obtained by using the same clause during re-extraction.
- 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.

3.3 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, and RDBMS do not guarantee tuple order, the tuple order in $\mathcal{W}$ is unimportant.)

The simplest resumption algorithm, called *Redo*, simply repeats the load. First $\mathcal{C}$ is deleted, and then for each extractor in the component DAG, the re-extraction procedure GetAllReordered() is invoked. *Redo* is shown in Figure 4.

Given component DAG, and $\mathcal{C}$ loaded in the warehouse 1. Delete $\mathcal{C}$. 2. For each extractor E in the component DAG   3. Call E.GetAllReordered()

Figure 4: *Redo* Algorithm

Although *Redo* is very simple, it still requires that the load be deterministic and that the original tuples be available. The following property, Same-set, is sufficient to enable *Redo*.

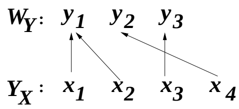

Figure 5: Safe Filtering of x_2

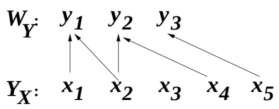

Figure 6: Unsafe Filtering of x_2

Definition 3.1 (**Same-set(**Y**)**) If Y is an extractor then Same-set(Y) = true if Y uses GetAll or GetAllReordered during resumption. Otherwise, Same-set(Y) = true if $\forall Y_X$: Same-set(X) and, given the same sets of input tuples, Y produces the same set of output tuples. □

While most transforms satisfy the Same-set property, there are some transforms that do not. For instance, a transform Y that adds a "ranking attribute" based on the position of the tuple in the input sequence does not satisfy this condition. However, to avoid producing non-deterministic values for the ranking attribute, a transform like Y is most likely preceded by a sorting transform. Thus, it is reasonable to assume that the same tuples are inserted into the warehouse when the same tuples are re-extracted because each transform produces the same tuples as output.

4 Properties for Resumption

Unlike *Redo*, an efficient resumption algorithm does not need to reprocess all of the tuples originally extracted from the sources. The *DR* resumption algorithm relies on the properties, attributes and keys declared when the component DAG is designed to avoid reprocessing some of the input tuples.

To illustrate, suppose that the sequence $\mathcal{W}_Y$ to be inserted into the warehouse is $[y_1y_2y_3]$ (see Figure 5) and $[x_1x_2x_3x_4]$ is the $\mathcal{Y}_X$ input sequence that yields the warehouse tuples. An edge $x_i \rightarrow y_j$ in Figure 5 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 stored in the warehouse. Clearly, it is *safe* to filter $\mathcal{Y}_X$ tuples that contribute only to $\mathcal{W}_Y$ tuples already in the warehouse, in this case, y_1. Thus in Figure 5, x_1 and x_2 can be filtered out. We need to be careful with y_1 contributors that also contribute to other $\mathcal{W}_Y$ tuples. For example, in Figure 6, $\{x_1, x_2\}$ again contribute to y_1, but we cannot filter out 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 properties, attributes and key attributes declared when the component DAG was designed.

In Section 4.1, we define the *map-to-one*, *suffix-safe*, *set-to-seq*, and *in-det-out* properties and derive "transitive" properties to answer Question (2). Then in Section 4.2, we use the key attributes to determine *identifying attributes* of the tuples to answer Question (1). In Section 6, we present a study that shows that the properties hold for many commercial transforms. In Section 4.3, we illustrate the properties through our working example.

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 $\mathcal{Y}_\mathcal{O}$ 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.[1]

[1]Given $\mathcal{T} = [t_1..t_n]$ and $\mathcal{S} = [s_1..s_k]$, $\mathcal{S}$ is a subsequence of $\mathcal{T}$ if there exists a strictly increasing sequence $[i_1..i_k]$ of indices of $\mathcal{T}$ such that for all $j = 1, 2, \ldots, k$, $t_{i_j} = s_j$ ([CLR92]).

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 can extend Definition 4.1 in a transitive fashion to define when a tuple contributes to a warehouse tuple. For instance, if a x_i contributes to y_j, which in turn contributes to a warehouse tuple z_k, then x_i contributes to z_k.

Notice that there may be tuples that do not contribute to any output tuple. For instance, if transform Y computes the sum of its input tuples and an input tuple t is $\langle\ 0\ \rangle$, then according to Definition 4.1, t does not contribute to the sum unless t is the only input tuple. Tuples like t that do not affect the output are called *inconsequential input tuples*, and are candidates for filtering.

Notice also that x_i is not considered to contribute to y_j, if y_j changes only its position in the output sequence $\mathcal{Y}_O$ when x_i is removed from the input sequence. In Appendix A, we discuss why the position of the output tuples is not considered in the definition of contributes.

Definition 4.1 does not consider transforms with non-monotonic input parameters. Informally, Y_X is non-monotonic if the number of output tuples of Y grows when the number of input tuples to Y_X is decreased. For instance, if Y is the difference transform $Y_{X1} - Y_{X2}$, Y_{X2} is non-monotonic. In this paper, we do not filter input tuples of a non-monotonic input parameter.

4.1 Safe Filtering

During resumption, a transform Y may not be required to produce all of its normal operation output $\mathcal{Y}_O$. Therefore, Y may not need to reprocess some of its input tuples, either. 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). A study presented in Section 6 confirms that input parameters of many transforms are map-to-one. For instance, the input parameters of selection, projection, union, aggregation and some join transforms 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 of the tuples in $\mathcal{Y}_O$ are not needed, then the corresponding tuples in $\mathcal{Y}_X$ that contribute to them can be safely filtered at resumption time. For example, in Figure 5, if the $\mathcal{Y}_O$ output tuples are the tuples being loaded into the warehouse, and tuples y_1 and y_2 are already committed in the warehouse, then the subset $\{x_1, x_2, x_4\}$ of the input tuples does not need to be processed and can be filtered from the Y_X input.

Subset-feasible(Y_X) is a transitive property that states that it is feasible to filter some subset of the Y_X input tuples. If there is a single path[2] from Y_X to the warehouse, Subset-feasible holds when all of the input parameters in the path are map-to-one. In this case, we can safely filter the Y_X tuples that contribute to some warehouse tuple for these Y_X tuples contribute to no other. Similarly,

[2]Formally, a path P in a component DAG is a sequence of edges where each pair of consecutive edges E_iE_j represents the input and output parameters Y_X and $Y_O = Z_Y$ of a transform Y. If P is composed of one edge, the edge must represent W_X, where X is the extractor that feeds the inserter W.

if there are multiple paths from Y_X to the warehouse, each input parameter along any path from Y_X to the warehouse must be map-to-one. If even one of the input parameters in the path(s) is not map-to-one, then we cannot filter any Y_X tuples because each Y_X tuple may contribute to tuples that are not yet in the warehouse.

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 $\forall Z_Y$: 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 likely to be suffix-safe (see Section 6).

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}))). If Contributors($\mathcal{Y}_X$, y_j) = [ ] or Contributors($\mathcal{Y}_X$, y_{j+1}) = [ ], then Y_X is not suffix-safe. □

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, tuples like x_3 that do not contribute to any output tuple can be filtered. Filtering such tuples is not possible using the map-to-one property.

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 $\forall Z_Y$: 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. We expect most transforms to satisfy this property.

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 X and all of the transforms preceding X are in-det-out, and the data extractors produce the same sequences at resumption time, then X will produce the same sequence, too. Hence, Y_X 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, if the sequence $[z_1 z_2 z_3 z_4]$ is produced by a sorting transform Z, then as long as Z processes the same set of tuples, $[z_1 z_2 z_3 z_4]$ is produced as output.

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 that holds if Y_X is guaranteed to receive the same sequence at resumption time. Same-seq(Y_X) is true if the transforms and input parameters that precede Y_X satisfy the in-det-out or set-to-seq property, respectively. Same-seq(Y_X) guarantees that Y_X receives the same input sequence. 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 GetAll 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 the bytes of a key attribute, 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 by matching attributes, 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. The input parameters of many join transforms also do not have hidden contributors. We show later in Section 6 that many commercial transforms have input parameters with no hidden contributors.

Property 4.5 (no-hidden-contributor(Y_X)) Given transform Y with input parameter Y_X, *no-hidden-contributors*(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)): ($Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$) $\Rightarrow$ ($x_i.a = y_j.a$). □

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. For now, we illustrate how the potential contributors are found.

Example 4.1 Consider the component DAG shown in Figure 7. The labels below the edges, e.g., Z_Y, identify the input parameter, and the labels above the edges give the attributes of the input tuples, e.g., Attrs(Z_Y) = [cd]. If Z_Y has no hidden contributors, then all of the $\mathcal{Z}_Y$ contributors to a warehouse tuple w_k, denoted S_k, match w_k on [cd] (i.e., Attrs(Z_Y) $\cap$ Attrs(Z_O)). If Y_X has no

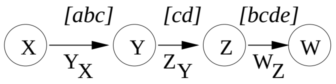

Figure 7: Example Component DAG

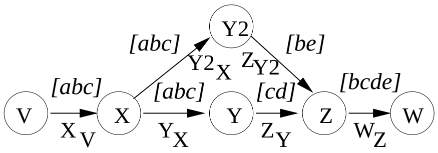

Figure 8: Component DAG with Replicated Outputs

hidden contributors, then all of the $\mathcal{Y}_X$ contributors to $z_i \in S_k$ match z_i on $[c]$ (i.e., Attrs(Y_X) $\cap$ Attrs(Y_O)). Since all of the tuples in S_k have the same c attribute (i.e., the c attribute of w_k), all of the $\mathcal{Y}_X$ tuples that contribute to w_k match w_k on $[c]$. Hence, all of the potential contributors of w_k in $\mathcal{Y}_X$ are the ones that match w_k on $[c]$. □

We call attributes that identify the $\mathcal{Y}_X$ potential contributors, the *candidate identifying attributes* or candidate attributes (CandAttrs) of Y_X. The formal definition of CandAttrs applies to an input parameter Y_X and a path P from Y_X to the warehouse.

Definition 4.5 (**CandAttrs(Y_X,P)**) Let P be a path from input parameter Y_X to the warehouse. There are three possibilities for CandAttrs(Y_X,P):

1. If Y is the warehouse inserter, then CandAttrs(Y_X, P) = Attrs(Y_X).
2. If $\neg$*no-hidden-contributors*(Y_X), then CandAttrs(Y_X, P) = [ ].
3. Else CandAttrs(Y_X, P) = CandAttrs(Z_Y, P') $\cap$ Attrs(Y_X), where $P = [Y_X Z_Y .. W_I]$, and P' is P excluding Y_X. □

In summary, CandAttrs(Y_X, P) is just the attributes that are present throughout the path P starting from Y_X, unless one of the input parameters in P has hidden contributors. If so, then CandAttrs(Y_X, P) is set to [ ] implying that all $\mathcal{Y}_X$ tuples are potential contributors.

Since the potential contributors identified by CandAttrs(Y_X, P) may include tuples that do not contribute to w_k, we would like to verify that all the potential contributors do contribute to w_k. To do so, we need to use key attributes. The *no-spurious-output* property is also useful but need not be satisfied to verify contributors. We define the no-spurious-output property to hold 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 (see Section 6), 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 in the next example how key attributes, candidate attributes, and the no-spurious-output property combine to determine the identifying attributes.

Example 4.2 Consider the component DAG shown in Figure 7. Note that CandAttrs(Y_X, P) = $[c]$ where $P = [Y_X Z_Y W_Z]$, assuming that Y_X, Z_Y, and W_Z have no hidden contributors. Now consider which attributes can be used as IdAttrs(Y_X). There are three possibilities.

1. IdAttrs(Y_X) = KeyAttrs(Y_X) if KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X, P) and both Y and Z satisfy the no-spurious-output property.
2. IdAttrs(Y_X) = KeyAttrs(W_Z) if KeyAttrs(W_Z) $\subseteq$ CandAttrs(Y_X, P).
3. IdAttrs(Y_X) = IdAttrs(Z_Y) if IdAttrs(Z_Y) $\subseteq$ CandAttrs(Y_X, P).

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

To illustrate the second possibility, suppose KeyAttrs(W_Z) = $[c]$. If $w_k.c = 1$, any $\mathcal{Y}_X$ tuple that contributes to w_k must have $c = 1$ since CandAttrs(Y_X, P) = $[c]$. All $\mathcal{Y}_X$ tuples with $c = 1$ must contribute to either w_k or to no warehouse tuples since c is the key of W_Z.

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

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

The previous example provides the intuition behind our definition of the identifying attributes of Y_X. The following definition gives the identifying attributes of Y_X along path P. If there is a single path P from Y_X to the warehouse, IdAttrs(Y_X) = IdAttrsPath(Y_X,P).

Definition 4.6 (IdAttrsPath(Y_X,P), IdAttrs(Y_X)) Let P be the only path from Y_X to the warehouse. There are three possibilities for IdAttrsPath(Y_X,P) (i.e., IdAttrs(Y_X)).

1. If (KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X, P) and $\forall Z_V \in P$: Z_V has no spurious output tuples), then (IdAttrsPath(Y_X, P) = KeyAttrs(Y_X)).
2. Otherwise, let $Z_V \in P$ but $Z_V \neq Y_X$. Let P' be the path from Z_V to the warehouse. If (IdAttrsPath(Z_V,P') $\neq$ [ ] and IdAttrsPath(Z_V,P') $\subseteq$ CandAttrs(Y_X, P)), then (IdAttrsPath(Y_X,P) = IdAttrsPath(Z_V,P')).
3. Otherwise IdAttrsPath(Y_X,P) = [ ].

□

Case (1) in Definition 4.6 uses the key attributes of Y_X as IdAttrs(Y_X). Case (2) in Definition 4.6 encompasses the second and third possibilities illustrated in Example 4.2. That is, for each input parameter in P, it checks if the IdAttrs of that input parameter can be used as IdAttrs(Y_X). Notice that there may be more than one input parameter in P whose identifying attributes can be used for IdAttrs(Y_X). We revisit this issue shortly.

We now modify IdAttrs to handle the general case where there are multiple paths from Y_X to the warehouse. The next example provides the intuition behind the generalization of IdAttrs.

Example 4.3 Consider the component DAG shown in Figure 8, where there are two paths from X_V to the warehouse: $P_1 = [X_V Y_X Z_Y W_Z]$ and $P_2 = [X_V Y2_X Z_{Y2} W_Z]$. We want to determine IdAttrs(X_V). Suppose IdAttrs(X_V, P_1) is $[b]$, while IdAttrs(X_V,P_2) is $[c]$. Then, given a warehouse tuple w_k, the $\mathcal{X}_V$ contributors along P_1 match w_k on $[b]$, while the $\mathcal{X}_V$ contributors along P_2 match w_k on $[c]$. Hence, the $\mathcal{X}_V$ contributors match w_k on $[b]$ or $[c]$, denoted $[b] \vee [c]$. □

Example 4.3 shows that if there are multiple paths, IdAttrs(Y_X) is the disjunction of the identifying attributes of the individual paths.

Definition 4.7 (IdAttrs(Y_X))

Let $\{P\}$ be the set of all paths from Y_X to the warehouse input parameter.
If $\exists P \in \{P\}$ such that IdAttrsPath(Y_X, P) = [ ], then IdAttrs(Y_X) = [ ].
Otherwise, IdAttrs(Y_X) = $\bigvee_{P\in\{P\}}$ IdAttrsPath(Y_X, P). □

Although we provide a general definition for IdAttrs(Y_X), in most cases there is a single path from Y_X to the warehouse. Even when there are multiple paths from Y_X to the warehouse, we can simplify IdAttrs(Y_X) as follows: Given IdAttrs(Y_X) = $A_1 \vee \ldots \vee A_n$, we eliminate A_i if $\exists A_j \subseteq A_i$, because any contributor identified by A_i is also identified by A_j.

This method for simplifying IdAttrs(Y_X) is also a guide for choosing the identifying attributes. When we developed IdAttrs(Y_X) for a single path P (Definition 4.6), we did not specify how to choose the input parameter along P whose IdAttrs will be used for IdAttrs(Y_X). It is best to choose an input parameter that appears on many paths from Y_X to the warehouse. The input parameter of the inserter W is a prime candidate since it appears on all paths from Y_X.

Transform	*Properties*	*Function computed by transform*
DT	in-det-out no-spurious-output	`select` $*$ `from` DT_{TRD} `where` $date \geq 12/1/97$ `and` $date \leq 12/31/97$
TV	in-det-out no-spurious-output	`select sum`($volume$) `as` $totalvol$ `from` TV_{DT}
PV	in-det-out no-spurious-output	`select` $PV_{PTE}.company, PV_{PTE}.pe,$ `sum`($PV_{DT}.volume$) $* 100/PV_{TV}.totalvol$ `as` $percentvol$ `from` $PV_{PTE}, PV_{DT}, PV_{TV}$ `where` $PV_{PTE}.company = PV_{DT}.company$ `and` $PV_{PTE}.pe \leq 4$ `group by` $PV_{PTE}.company, PV_{PTE}.pe$

Table 1: Properties and Functions of Transforms.

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	[ ]	-
TV_{DT}	[date,company,volume]	[date,company]	map-to-one suffix-safe set-to-seq	[ ]	Prefix-feasible
PV_{PTE}	[company,pe]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible
PV_{DT}	[date,company,volume]	[date,company]	map-to-one set-to-seq	[company]	Subset-feasible
PV_{TV}	[totalvol]	[totalvol]	suffix-safe set-to-seq	[ ]	Prefix-feasible
W_{PV}	[company,pe,percentvol]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible

Table 2: Declared and Inferred Properties of Input Parameters.

4.3 The Trades Example Revisited

We now return to our main example, shown in Figure 3, and illustrate the properties satisfied by the input parameters and transforms.

Table 1 shows the functions computed by the transforms. Although these function definitions cannot be used by the resumption algorithms, we include them here to help explain why the properties hold. We show SQL functions for simplicity even though transforms often perform functions that cannot be written in SQL. Table 1 also shows that all three transforms are declared to be in-det-out since they produce the same output sequence given the same input sequences.

The first four columns of Table 2 show the attributes, keys, and properties declared for each input parameter when the component DAG is designed. 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 1997. Therefore, DT_{TRD} is suffix-safe, since DT outputs tuples in the input tuple order. It is map-to-one, since 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 TV reads all of its input before producing one output tuple. TV_{DT} is trivially map-to-one, suffix-safe, and set-to-seq.

Transform PV reads each tuple in PV_{PTE} and if its pe attribute is ≤ 4, it finds all of the trade tuples for the same company in PV_{DT}, which are probably not in order by company. It computes the percent of the total trade volume using the trade tuples and PV_{TV} and outputs a tuple. Then it processes the next tuple in PV_{PTE}. PV_{PTE} is map-to-one since each tuple contributes to zero or one output tuple, depending on its value for the attribute pe. It is not set-to-seq for the same reason it is suffix-safe: PV processes tuples from PV_{PTE} one at a time, in order. PV_{DT} is map-to-one since each trade tuple contributes to the percent volume tuple of only one company. However, PV_{DT} is not suffix-safe, e.g., the trade tuple needed to join with the first tuple in PV_{PTE} may be the last tuple in PV_{DT}. Similarly, it is set-to-seq because the order of trades tuples is not relevant to PV. PV_{TV} is not map-to-one since the lone PV_{TV} input tuple containing the total volume contributes to all of the output tuples. PV_{TV} is trivially suffix-safe and set-to-seq.

Finally, since the warehouse inserter simply stores its input tuples in order, W_{PV} 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. We assume that none of the input parameters have hidden contributors. The identifying attribute of W_{PV}, PV_{DT}, and PV_{PTE} is [$company$] because it is the key of W_{PV}. Since none of the attributes of PV_{TV} are preserved in the warehouse, we cannot possibly identify the contributing $\mathcal{PV}_{TV}$ tuples, and IdAttrs(PV_{TV}) is set to [ ]. As a result, IdAttrs(TV_{DT}) = IdAttrs(DT_{TRD}) = [ ]. The transitive properties (e.g., Subset-feasible) are computed 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.

4.4 Practical Issues

The properties that we have introduced hold in many cases. In Section 6, we present a thorough study of a commercial load package to support this claim. The properties are also fairly simple. In fact, some commercial load packages [Sag98] already declare whether some of the properties (e.g., suffix-safe) hold for their transforms. Even if the properties are not declared, they can often be deduced easily from the transform specifications or manuals. Moreover, the properties focus on

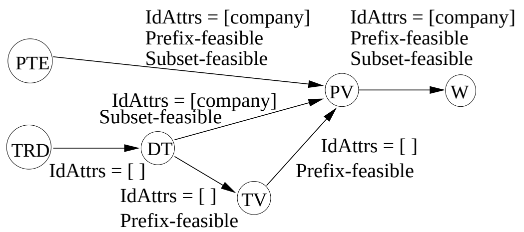

Figure 9: Identifying Attributes and Transitive Properties

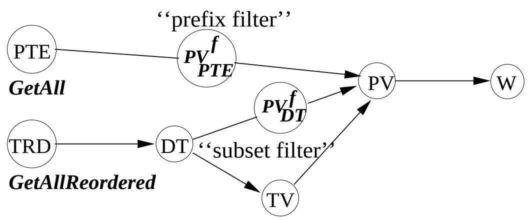

Figure 10: Re-extraction Procedures and Filters Assigned

a single transform and not the whole component DAG, which makes them easy to grasp. (The transitive properties are derived by the *DR* algorithm.)

5 The *DR* Resumption Algorithm

We now present the *DR* resumption algorithm, which uses the properties developed in the previous section. *DR* is actually composed of two algorithms, *Design* and *Resume*, hence the name. After a component DAG G is designed, *Design* constructs a component DAG G' that *Resume* will employ to resume any failed warehouse load that used G. The component DAG G' is the same as G except for the following differences.

1. Re-extraction procedures are assigned to the extractors in G'.
2. Filters are assigned to some of the input parameters in G'.

The component DAG G' is constructed by *Design* based solely on the attributes, keys, and properties declared for 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*

To illustrate the overall operation of *DR*, we return to our running example (Figure 3). After this illustration, we cover *DR* in more detail.

Algorithm *Design* of *DR* first computes the Subset-feasible and Prefix-feasible transitive properties, as well as the IdAttrs of each input parameter. We computed these transitive properties and identifying attributes in Section 4.3, and the results are shown in Figure 9.

Design then constructs G' by first assigning re-extraction procedures to extractors based on the computed properties and identifying attributes. Since IdAttrs(PV_{PTE}) = [$company$], it is possible to identify source PE tuples that contribute to tuples in the warehouse based on the $company$ attribute. Since Prefix-feasible(PV_{PTE}) holds, *DR* can assign GetSuffix to PTE to avoid re-extracting all the PE tuples over again. Also, since Subset-feasible(PV_{PTE}) holds, *DR* can alternatively assign GetSubset to PTE to avoid re-extracting all the PE tuples. Suppose PTE supports neither GetSuffix nor GetSubset, so GetAll is assigned to PTE instead.

IdAttrs(DT_{TRD}) is empty, implying that it is not possible to identify the Trades tuples that contribute to warehouse tuples. Hence, assuming TRD does not support GetAll, only the re-extraction procedure GetAllReordered can be assigned to TRD.

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 input parameters. As a result, the input parameters are processed in topological order because the Same-seq property of an input parameter depends on the Same-seq properties of previous input parameters.

1. Same-seq(DT_{TRD}) does not hold because TRD is assigned GetAllReordered, so it is not possible to filter a prefix of the DT_{TRD} input sequence. Furthermore, since DT_{TRD} is not Subset-feasible, a subset filter cannot be assigned.
2. Same-seq(PV_{PTE}) holds because PTE is assigned GetAll. Therefore, PV_{PTE} is both Prefix-feasible and Same-seq, so it is possible to filter a prefix of the PV_{PTE} input sequence. Furthermore, we can identify the contributors to the warehouse tuples based on IdAttrs(PV_{PTE}) = [*company*]. Thus, a filter, denoted PV^f_{PTE}, that removes a prefix of the PV_{PTE} input sequence is assigned to PV_{PTE}. When a failed load is resumed, PV^f_{PTE} removes the prefix of the PV_{PTE} input sequence that ends with the tuple whose *company* attribute matches the last warehouse tuple.
3. TV_{DT} is Prefix-feasible but we cannot identify the contributors of the warehouse tuples since IdAttrs(TV_{DT}) = [ ]. Furthermore Same-seq(TV_{DT}) does not hold since Same-seq(DT_{TRD}) does not hold. No filter is assigned to TV_{DT}.
4. PV_{DT} is Subset-feasible and IdAttrs(PV_{DT}) = [*company*], so a subset filter PV^f_{DT} is assigned to PV_{DT}. Same-seq(PV_{DT}) does not hold, but the subset filter PV^f_{DT} does not require it. When a failed load is resumed, PV^f_{DT} removes all tuples in the PV_{DT} sequence whose *company* attribute value matches some warehouse tuple.
5. IdAttrs(PV_{TV}) is [ ], so no filter is assigned to PV_{TV}. Note that Same-seq(PV_{TV}) holds since TV_{DT} is set-to-seq.
6. Finally, Same-seq(W_{PV}) cannot hold since the filters assigned to PV_{PTE} and PV_{DT} make it impossible for W_{PV} to receive the same sequence. A subset filter can be assigned to W_{PV} since W_{PV} is Subset-feasible. However, *Design* determines that this filter is redundant with the previous filters. Therefore, no filter is assigned to W_{PV}.

The component DAG G' constructed from G is shown in Figure 10. Note that G' is constructed using just two "passes" over G: a backward pass to compute IdAttrs, Prefix-feasible, Subset-feasible, and a forward pass to compute Same-seq. Hence, the time to construct G', which is in the order of seconds or minutes, is negligible compared to the time to design and debug G, which is in the order of days or weeks ([Inc]). Algorithm *Design* is now done. Until a failed load by G is resumed, G' is not used.

Suppose that a load using G fails, and the tuple sequence that made it into the warehouse is

$$\mathcal{C} \;=\; [\ \langle AAA, 3, 0.25\rangle\ \langle INTC, 2, 0.98\rangle\ \langle MSN, 4, 0.456\rangle\ ],$$

where the three attributes are *company*, *pe*, and *percentvol*, respectively. Based on $\mathcal{C}$, *Resume* instantiates the filters and re-extraction procedures (i.e., GetSuffix, GetSubset) that are sensitive to $\mathcal{C}$. Since only GetAll and GetAllReordered are assigned in our example, only the filters are affected by $\mathcal{C}$.

Subset filter PV^f_{DT} is instantiated to remove any PV_{DT} input tuple whose company is either AAA, $INTC$ or MSN. It is safe to filter these tuples since it is guaranteed that they contribute to at most one warehouse tuple (i.e., Subset-feasible(PV_{DT})), and that tuple is in $\mathcal{C}$. Similarly, prefix filter PV^f_{PTE} is instantiated to remove the prefix of its PV_{PTE} input that ends with the tuple whose *company* attribute is MSN. It is safe to filter these tuples since it is guaranteed that PV has processed through the MSN tuple of PV_{PTE} (i.e., Prefix-feasible(PV_{PTE})). Note that the tuples before the MSN tuple may include ones that do not contribute to any warehouse tuple (i.e., because their *pe* attribute is too high). Once the filters and re-extraction procedures are instantiated, the warehouse 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 PV and W. Had the load failed with a longer warehouse tuple sequence $\mathcal{C}$, the filters would have been instantiated appropriately by DR to filter more input tuples.

We conclude the example by contrasting the recovery performed by DR with other methods.

- Unlike *Redo*, DR avoids re-processing many of the input tuples using filters PV^f_{DT} and PV^f_{PVE}. Also, had the extractors PTE and TRD supported GetSubset or GetSuffix, DR could have even avoided re-extracting tuples from the sources.
- DR avoids re-processing many input tuples without having to identify batches. Recall that for our example component DAG (Figure 9), batches cannot be formed due to the *TotalVolume* (TV) transform. Since batches cannot be formed, a recovery algorithm based on batching input tuples would redo the entire warehouse load.
- During normal operation, the designed component DAG G (Figure 9) is used. No normal operation overhead is incurred unlike recovery algorithms based on savepoints or snapshots. Again, the time it takes to construct G' from G is very small compared to the time it takes to design and debug G. Furthermore, this overhead occurs when G is designed, and does not occur during the normal operation of the load.

5.2 Filters

In the previous example, we mentioned subset filters and prefix filters. More specifically, 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 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 the clean-prefix filter. DP discards tuples from its input sequence until it finds a tuple t that matches s on A. All tuples starting with t are output by DP. We use DP on Y_X when Y_X is Prefix-feasible, and Same-seq, and IdAttrs(Y_X) is not empty. In this case, all input tuples preceding the contributors of Last($\mathcal{C}$) can be safely filtered. So CP is instantiated as $CP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$.

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 does not match any $\mathcal{S}$ tuple

Algorithm 5.1 *AssignFilter*
Input: Component DAGs G, G'; input parameter Y_X
Output: Input parameter Y_X in G' is assigned a filter whenever possible

1. If Prefix-feasible(Y_X) and Subset-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   2. Insert $Y_X^f = CP$[Last($\mathcal{C}$),IdAttrs(Y_X)] between Y and X in G'
3. Else If Prefix-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   4. Insert $Y_X^f = DP$[Last($\mathcal{C}$), IdAttrs(Y_X)] between Y and X in G'
5. Else if Subset-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   6. Insert $Y_X^f = CS$[$\mathcal{C}$, IdAttrs(Y_X)] between Y and X in G'
7. Else if Prefix-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   8. Insert $Y_X^f = DS$[$\mathcal{C}$, IdAttrs(Y_X)] between Y and X in G'

Figure 11: Assigning Input Parameter Filters

on the A attributes, then t is output. Otherwise, t is discarded. 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, $DS[\mathcal{C},\text{IdAttrs}(Y_X)]$, is a slight modification to the clean-subset filter. DP is assigned to Y_X when Y_X is Prefix-feasible and IdAttrs(Y_X) is not empty. Unlike CS, DS removes a suffix $\mathcal{C}_s$ of $\mathcal{C}$ before performing the anti-semijoin. $\mathcal{C}_s$ contains all the tuples that share Y_X contributors with Last($\mathcal{C}$). This suffix can be obtained easily by matching all the $\mathcal{C}$ tuples with the Last($\mathcal{C}$) tuple on IdAttrs(Y_X). After $\mathcal{C}_s$ is obtained, a prefix of $\mathcal{C}$, denoted $\mathcal{C}_p$, is obtained by removing $\mathcal{C}_s$ from $\mathcal{C}$. $\mathcal{C}_s$ is removed since we cannot filter the contributors to $\mathcal{C}_s$ because Y_X is not required to be Subset-feasible. DP then acts like the clean-subset filter $CS[\mathcal{C}_p,\text{IdAttrs}(Y_X)]$.

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 we have introduced are as follows.

$$CP \succ DP \succ DS, \quad CP \succ CS \succ DS$$

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.

The procedure *AssignFilter* is shown in Figure 11. Observe that *AssignFilter* assigns a filter to Y_X whenever possible. Since some of these filters may be redundant with previous filters, *Design* uses a subsequent procedure to remove redundant filters.

So far, we have implicitly assumed in our discussion that IdAttrs is a single attribute set, when in general it could be a disjunction of attribute sets. While it is usually the case that IdAttrs is a single attribute set (as in our working Trades example), there may be cases where it is not. In Appendix B, we show that only the implementation of the filters (and re-extraction procedures) need to be changed when IdAttrs is a disjunction of attribute sets. We also provide the minor changes to the filter implementation required. The rest of the *DR* algorithm, like deciding what type of filter to assign, is unaffected.

5.3 Re-extraction Procedures

We now define the re-extraction procedures formally. From these definitions, it is clear that the re-extraction procedures are very similar to the filters. In particular, the re-extraction procedures GetSuffix and GetSubset perform the same processing as the CP and CS filters, respectively. Furthermore, we introduce the re-extraction procedures GetDirtySuffix and GetDirtySubset that correspond to the DP and DS filters.

Definition 5.1 (**Re-extraction procedures for resumption**)
GetAll() = $\mathcal{E}_O$, where $\mathcal{E}_O$ was the the output of E during normal operation.
GetAllReordered() = $\mathcal{T}$: $\mathcal{T}$ 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) = $\mathcal{T}$: $DS[\mathcal{S}, A] = \mathcal{T}$. □

Since the re-extraction procedures and filters perform similar processing, it is not surprising that the procedure *AssignReextraction* is similar to *AssignFilter*. To illustrate, consider an extractor E and a component Y that receives E's output. If Prefix-feasible(Y_E), Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ], then we can assign a clean-prefix filter CP to Y_E. However, this filter can be "pushed" to E if E supports GetSuffix. Similarly, the other parts of *AssignReextraction* tries to push the remaining filter types from Y_E to E. In Section 6, we show experimentally the benefits of pushing the filtering to the extractors. If no filter can be pushed to an extractor E, either GetAll or GetAllReordered is assigned to it. The detailed listing of *AssignReextraction* is in Appendix B.

5.4 The *Design* and *Resume* Algorithms

Algorithm *Design* of *DR* (Algorithm 5.2, Figure 12) starts by computing the IdAttrs and the Prefix-feasible and Subset-feasible transitive properties of each input parameter Y_X in the given component DAG G. The input parameters are processed in reverse topological order because all of the above properties of Y_X depend on the properties of subsequent input parameters (e.g., Z_Y).

Then *Design* calls *AssignReextraction* to assign re-extraction procedures to each extractor in G'. Next, *Design* computes the Same-seq property and calls *AssignFilter* (Figure 11) to assign filters to each input parameter in G'. Since the Same-seq property of Y_X depends on the Same-seq properties of previous input parameters, the input parameters are processed in topological order. Note that Same-seq(Y_X) is set to false if a filter is assigned to Y_X, because the filter ensures that Y_X does not receive the same input sequence as it did during normal operation. Redundant filters are removed and then G' is saved persistently.

In case of failure, *Resume* of *DR* (Algorithm 5.3, Figure 12) 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. Note that *Resume* can be invoked multiple times on the same G', while *Design* only needs to be called once, at design time, regardless of the number of failures.

We now discuss how redundant filters are detected by *Design*. We say a filter Y_X^f is redundant if Y_X^f is guaranteed not to discard any tuples. Given a path P in G, with V_U preceding Y_X in P, Y_X^f in G' is redundant if there is a filter V_U^f in G' and the following two conditions hold:

Algorithm 5.2 *Design*
Input: Component DAG G
Output: Component DAG G'

1. $G' \leftarrow G$ // copy 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
   4. *AssignReextraction*(G,G',E)
5. For each input parameter Y_X in topological order
   6. Compute Same-seq(Y_X)
   7. *AssignFilter*(G,G',Y_X)
   8. If Y_X is assigned a filter, set Same-seq(Y_X) to false.
9. *RemoveRedundantFilters*(G, G')
10. Save G' persistently and return G'

Algorithm 5.3 *Resume*
Input: Component DAG G'
Side Effect: Resumes failed warehouse load using G
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'
   3. Invoke re-extraction procedure assigned to E

Figure 12: *DR* Algorithm

1. V_U^f is of filter type f (e.g., CP) and Y_X^f is of filter type g (e.g., CS) and $f \succ g$ or $f = g$.
2. IdAttrs(V_U) $\subseteq$ IdAttrs(Y_X).

Once Y_X^f is detected as redundant, it is removed from G'. A brute force way to detect redundant filters is to consider each path in G' and check the above conditions. An efficient implementation of *RemoveRedundantFilters* is in Appendix B.

We now analyze the complexity of *DR*. Let n be the number of nodes in G. Steps 2–8 of *Design* produce a topological ordering of the nodes in G and then traverse it. They take $O(n^2)$ time. Detecting redundant filters in step 9 also takes $O(n^3)$ time (see Appendix B). *Resume* instantiates at most $O(n^2)$ filters. Usually many fewer than $O(n^2)$ filters are created. Furthermore, we show in our experiments (Section 6) that even adding a single filter can dramatically improve performance. Subset filters can be instantiated in $O(|\mathcal{C}|)$ time, where $|\mathcal{C}|$ is the number of warehouse tuples. Prefix filters are instantiated in $O(1)$ time (with appropriate indices on warehouse tables). Therefore, *DR* runs in $O(n^2 \cdot |\mathcal{C}| + n^3)$ time.

5.5 Correctness of *DR*

A correct load resumption algorithm produces the same set of tuples in the warehouse as the original load would have, had there been no failures. By this definition, *DR* is correct. *DR* only filters tuples that are not needed to produce subsequent warehouse tuples. Furthermore, no warehouse tuple in $\mathcal{C}$ is reproduced.

DR only filters unneeded tuples because it relies on the properties defined in Section 4. For instance, if Subset-feasible(Y_X) holds, then *DR* can safely filter some Y_X tuples, knowing that those tuples only contribute to a warehouse tuple already in $\mathcal{C}$. *DR* ensures that none of the $\mathcal{C}$ tuples is reproduced by guaranteeing that a CP or CS filter is assigned. Since a clean filter removes all of the contributors to $\mathcal{C}$ tuples, none of the $\mathcal{C}$ tuples are reproduced. Since the input parameter of W is guaranteed to be Subset-feasible and have non-empty IdAttrs, *DR* can always assign it a CS filter (if no other filter assignment is possible).

6 Experiments

In this section, we present our experiments that compares *DR* to other recovery algorithms. We also show that the properties on which *DR* relies are quite common.

6.1 Study of Transform Properties

Sagent's Data Mart 3.0 is commercial software for constructing component DAGs for warehouse creation and maintenance. It provides 5 types of warehouse inserters, 3 types of extractors, and 19 transforms. The software also allows users to create their own transforms.

All three extractors support GetAll and GetAllReordered, but only the "SQL" Extractor supports GetSuffix, GetDirtySuffix, GetSubset and GetDirtySubset. Of the 19 transforms, 15 have one input parameter, and the other 4 have two input parameters, for a total of 23 input parameters. Figure 13 shows a summary of the properties that hold for the 19 transforms and the 23 input parameters. The 19 transforms include Sagent's implementations of conventional operations used in databases, such as selection, projection, union, aggregation, and join.

- 100% (19 out of 19) of the transforms are in-det-out.
- 95% (18 out of 19) of the transforms have no spurious output.
- 91% (21 out of 23) of the input parameters are map-to-one.
- 78% (18 out of 23) of the input parameters are suffix-safe.
- 17% (4 out of 23) of the input parameters are set-to-seq (i.e., perform sorting).
- 100% (23 out of 23) of the input parameters have no hidden contributors.

Figure 13: Properties of Sagent Transforms and Input Parameters

Some of these properties, like suffix-safe, are actually declared by Sagent. Other properties were deduced easily from the Sagent manuals that specify the transforms. The statistics in Figure 13 imply that the transitive properties Subset-feasible (due to map-to-one), Prefix-feasible (due to suffix-safe) and Same-seq (due to in-det-out and set-to-seq) hold for many component DAG scenarios.

The only conventional database operation that Sagent does not implement is the difference operation: $Z_Y - Z_X$. The table below summarizes the properties satisfied by a difference transform Z and its input parameters. The table implies that it may be possible to filter Z_Y tuples. However, since Z_X is neither map-to-one nor suffix-safe, it is not possible to filter Z_X tuples.

Z properties	Z_Y properties	Z_X properties
in-det-out	map-to-one suffix-safe no-hidden-contributor	set-to-seq

Table 3: Properties of a Difference Transform.

6.2 Resumption Time Comparison

We performed experiments using Sagent's Data Mart 3.0 to construct various component DAGs. The software ran on a Dell XPS D300 with a Pentium II 300 MHz processor and 64 MB of RAM.

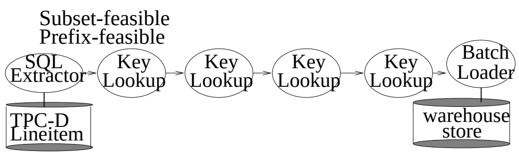

Figure 14: Fact Table Creation DAG

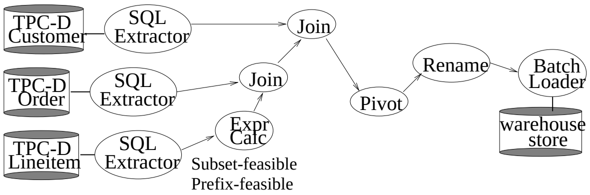

Figure 15: TPC-D View Creation DAG

We designed three types of component DAGs. One type of component DAG loads *dimension tables*, e.g., the *Customer* and *Supplier* TPC-D tables [Com]. Dimension tables typically store data about entities like customers. Another type of component DAG loads *fact tables*, e.g., the *Order* and *Lineitem* TPC-D tables. Fact tables typically store transactional data. The last type of component DAG loads materialized views that contain the answers to queries, e.g., TPC-D queries. Since the results of the dimension and fact table scenarios were very similar, we only present results for the fact table and the TPC-D materialized view scenarios. The component DAGs for loading the TPC-D fact table *Lineitem*, and the materialized view for the TPC-D query *Q3* are shown in Figures 14 and 15 respectively. Query *Q3*, the "shipping priority query," joins 3 tables and performs a GROUP BY and a SUM of revenue estimates.

Experiment 1: In the first experiment, we compared the resumption times of *DR*, *Redo*, and the algorithm used by Informatica ([Inf]), denoted *Inf*, for the *Lineitem* DAG (Figure 14). Recall that *Inf* filters the input to the inserter "*BatchLoader*" based on the warehouse tuples. No other filters are employed by *Inf*. The three algorithms compared impose no overhead during normal operation but can handle complex workflows. That is, all the algorithms are in the lower right quadrant of Figure 2 (Section 1). Furthermore, we studied "variants" of *DR* by assuming different properties for the component DAG.

Variant DR_{src}: DR_{src} pushes filtering to the re-extraction procedure at the source. In Figure 14, the transform properties show that $KeyLookup_{SQLExtractor}$ is both Prefix-feasible and Subset-feasible, and the extractor for *Lineitem* supports GetSuffix. Therefore, DR_{src} assigns GetSuffix to the *Lineitem* extractor.

Variant DR_{pre}: DR_{pre} assigns a prefix filter immediately after the *Lineitem* source. In Figure 14, DR_{pre} places a clean-prefix filter between the *Lineitem* extractor and *KeyLookup*. This component DAG will be constructed when the *Lineitem* extractor does not support GetSuffix.

Variant DR_{sub}: DR_{sub} assigns a subset filter immediately after the *Lineitem* source. In Figure 14, DR_{sub} assigns a clean-subset filter to $KeyLookup_{SQLExtractor}$.

We compared *Redo*, *Inf* and the variants of *DR* under various failure scenarios. More specifically, we investigated scenarios where 0%, 20%, 40%, 60%, 80% and 95% of the warehouse table is loaded when the failure occurs. For example, since *Lineitem* has 60,000 tuples (i.e., 0.01 TPC-D scaling), we investigated failures that occurred after loading 0 to 57,000 tuples. A low scaling factor was used so that the experiment can be repeated numerous times.

The results are shown in Figure 16, which plots the resumption time of *Inf*, *Redo*, DR_{src}, DR_{pre} and DR_{sub} as more tuples are loaded into the warehouse before the failure. As expected, DR_{src}, DR_{pre} and DR_{sub} all perform better than *Redo* once 20% (or more) of the *Lineitem* tuples reach the warehouse. For instance, when *Lineitem* is 95% loaded, DR_{src} resumes the load 10.4 times

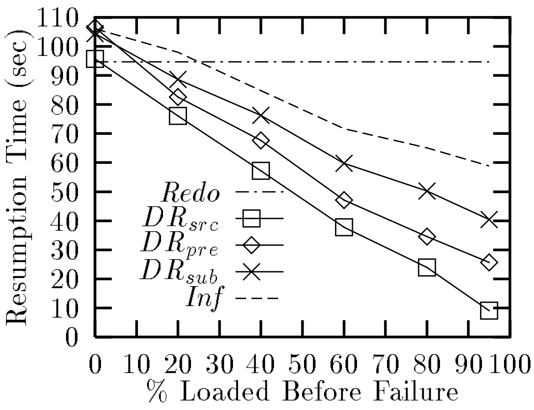

Figure 16: Resumption Time (*Lineitem*)

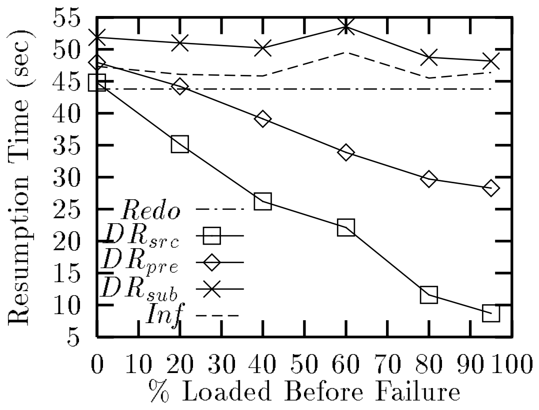

Figure 17: Resumption Time (*Q3*)

faster than *Redo*, DR_{pre} resumes the load 3.68 times faster, and DR_{sub} resumes the load 2.35 times faster. The variants of *DR* also resume the load significantly faster than *Inf*. For instance, when *Lineitem* is 95% loaded, DR_{src} resumes the load 6.46 times faster than *Inf*, DR_{pre} resumes the load 2.28 times faster, and DR_{sub} resumes the load 1.45 times faster. On the other hand, when none of the *Lineitem* tuples reach the warehouse before the failure, *Inf*, DR_{sub}, and DR_{pre} perform worse than *Redo* because of the overhead of the filters they use. More specifically, when *Lineitem* is 0% loaded, *Redo* is 1.12 times faster than DR_{pre}, 1.10 times faster than DR_{sub}, and 1.12 times faster than *Inf*. The overhead of the filters can be minimized by improving their implementation. DR_{src} which pushes the filtering to the *Lineitem* source, is almost as fast as *Redo* when the warehouse table is 0% loaded. Preliminary experiments using 1.0 TPC-D scaling show very similar relative improvements by the *DR* variants over *Redo* when enough *Lineitem* tuples are loaded.

Among the three *DR* variants, DR_{src} performs the best since it filters the tuples the earliest. DR_{sub} performs worse than DR_{pre} because of the overhead of the anti-semijoin operation employed by DR_{sub}'s subset filters. Furthermore, the next experiment will show that DR_{pre} filters more tuples than DR_{sub}.

Experiment 2: The second experiment is similar to the first but considers the *Q3* DAG (Figure 15). The results are shown in Figure 17. As in the first experiment, DR_{pre} and DR_{src} perform better than *Redo* once 20% (or more) of the warehouse table tuples is loaded. For instance, when the warehouse table is 95% loaded, DR_{src} is 5.03 times faster than *Redo*, and DR_{pre} is 1.55 times faster than *Redo*. However, DR_{sub} and *Inf* perform worse than *Redo* regardless of how many tuples are loaded. For instance, *Redo* is 1.22 times faster than DR_{sub} when the warehouse table is 60% loaded. The reason why DR_{sub} and *Inf* do not perform well is that query *Q3* 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 DR_{sub} do not remove enough tuples to compensate for the cost of the filter. Similarly, the filter used by *Inf* removes tuples based only on the warehouse tuples. Just like DR_{sub}, *Inf* does not filter many tuples.

Experiment 3: In the third experiment, we examined the normal operation overhead of a recovery algorithm that is based on savepoints. Such an algorithm is a representative of the algorithms in the upper right quadrant of Figure 2 (Section 1). We again considered the *Lineitem* and *Q3* component DAGs. For the former component DAG, we introduced 1 to 3 savepoints. For instance, the first savepoint records the result of the first "*KeyLookup*" transform. The results are shown in Figure 18. Without savepoints, the *Lineitem* table is loaded in 94.7 seconds. As the table shows, one savepoint makes the normal operation load 1.76 times slower, two savepoints make the normal load 2.6 times slower, and three savepoints make the normal load 3.3 times slower. On the other hand, the algorithms compared in the first two experiments (e.g., *DR*) have no normal operation overhead, and do not increase the load time.

For the *Q3* DAG, we also introduced 1 to 3 savepoints. The first savepoint records the result of the first "*Join*" transform, the second records the result of the second "*Join*" transform, and the third records the result of the "*Pivot*" and "*Rename*" transform. The normal operation overhead of the savepoints is tolerable for this component DAG. Even with three savepoints, the normal operation load is only about 1.08 times slower. The reason why the savepoints do not incur much overhead is that the "*Join*" transforms are very selective. Hence, only few tuples are recorded in the savepoints. More specifically, the first savepoint records 1344 tuples, the second records 285 tuples, and the third records 103 tuples.

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

Figure 18: Savepoint Overhead (*Lineitem*)

# Savepoints	Load Time (s)	% Increase Load Time
0	43.8	0%
1	46.1	5.3%
2	46.9	7.1%
3	47.2	7.8%

Figure 19: Savepoint Overhead (*Q3*)

Experiment 4: In the fourth experiment, we compared the resumption time of *DR* against an algorithm based on savepoints, denoted *Save*. We compared the two algorithms under various failure scenarios. For instance, for *DR* we would load the warehouse using the *Lineitem* DAG, and stop the load after t_{fail} seconds. To simulate various failure scenarios, we would vary t_{fail}. We then resumed the load using *DR* and recorded the resumption time. For *Save*, we would load the warehouse using the same *Lineitem* DAG, but with savepoints. We also stop the load after t_{fail} seconds. We then resumed the load using any completed savepoints. We again considered the *Lineitem* and *Q3* DAGs. In the case of *Save*, we used two savepoints for each component DAG.

The result for the *Lineitem* DAG is shown in Figure 20 which plots the resumption time of *DR* and *Save* as t_{fail} is increased. The graph shows that *Save*'s resumption time improves in discrete steps. For instance, when $t_{fail} < 79$ seconds, the first savepoint has not completed and cannot be used. Once $t_{fail} > 79$ seconds, the first savepoint can be used to make resumption more efficient. For the *Lineitem* DAG, *DR* is more efficient than *Save* in resuming the load. This is because the warehouse table is populated early in the load, and *DR* can use the warehouse table tuples to make resumption efficient.

The result for the *Q3* DAG is shown in Figure 21. Again, *Save*'s resumption time improves in discrete steps based on the completion of the savepoints. For this DAG, *DR*'s resumption time does not improve until t_{fail} is near 43 seconds (when the load completes). This is because the second "*Join*" transform takes in excess of 30 seconds to produce its first output tuples. As a result, the warehouse table is not populated until the load time is near 43 seconds. For this DAG, *Save* is slightly more efficient than *DR* in resuming the load for many values of t_{fail}. Unfortunately, both *Save* and *DR* do not perform well.

To improve the resumption performance, a hybrid algorithm that combines the features of *Save* and *DR* can be employed. The two savepoints employed by *Save* essentially partition the *Q3* DAG into three "sub-DAGs." However, *Save* does not make use of incomplete savepoints to improve resumption. On the other hand, *DR* can be used to treat an incomplete savepoint and the "sub-DAG" that produced it as if it was a warehouse table being loaded by a component DAG. The performance of the hybrid algorithm, denoted *DR-Save*, is plotted in Figure 21. For most values of t_{fail}, *DR-Save* is better than either *Save* or *DR*.

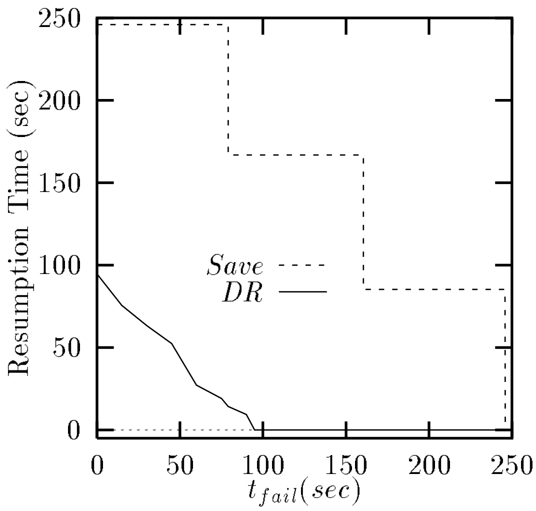

Figure 20: *Save* vs. *DR* (*Lineitem*)

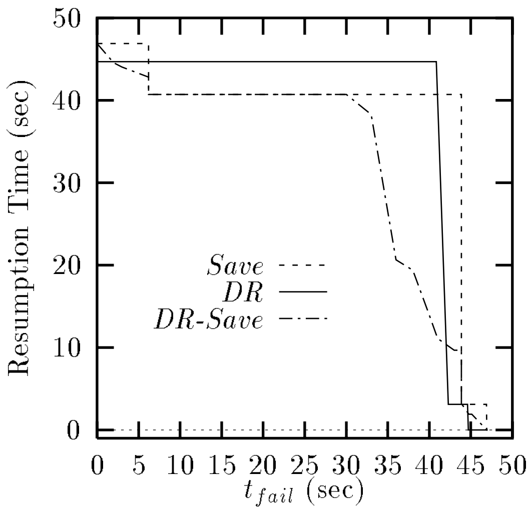

Figure 21: *Save* vs. *DR* (*Q3*)

Experiment 5: In the fifth experiment, we examined the normal operation overhead of a recovery algorithm that is based on batching. Such an algorithm is a representative of the algorithms in the lower left quadrant of Figure 2 (Section 1). We again considered the *Lineitem* and *Q3* component DAGs. For the former component DAG, we loaded *Lineitem* in three input batches. The results are shown in Figure 22. The table shows that batching results in a significant overhead especially when 4 or more batches are used.

For the *Q3* DAG, we also loaded the target table using three input batches. The results are shown in Figure 23. Again, the table shows that batching results in a significant overhead especially when 4 or more batches are used. Hence, when it is possible to divide the input into batches, one must be careful as to how many batches should be formed. A high number of batches results in significant normal operation overhead. On the other hand, a low number of batches results in a longer (average) resumption time

Experiment 6: In the sixth experiment, we compared the resumption time of *DR* against an algorithm based on batching, denoted *Batch*. The setup of this experiment is similar to the setup in Experiment 4. That is, for *DR* we would load the warehouse using the designed DAG and stop the load after t_{fail} seconds. We then measure the resumption time of *DR*. For *Batch*, we would load the warehouse by processing the input batches in sequence. For *Batch*, we used three input batches so that the normal operation overhead is tolerable. We then measure the resumption time

# Batches	Load Time (s)	% Increase Load Time
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%

Figure 22: Batching Overhead (*Lineitem*)

# Batches	Load Time (s)	% Increase Load Time
1	43.8	0%
2	44.6	1.8%
3	44.9	2.5%
4	49.1	12.1%
5	54.2	23.7%
10	76.2	74.0%

Figure 23: Batching Overhead (*Q3*)

of *Batch* based on the input batches that have been processed completely.

The result for the *Lineitem* DAG is shown in Figure 24 which plots the resumption time of *DR* and *Batch* as t_{fail} is increased. The graph shows that *Batch*'s resumption time improves in discrete steps. For instance, when $t_{fail} < 36$ seconds, the first input batch has not been processed completely. During resumption, the output based on the first input batch is discarded, and the first input batched is reprocessed. Once $t_{fail} > 36$ seconds, the first input batch has been processed completely and does not need to be reprocessed during resumption. For the *Lineitem* DAG, *DR* is surprisingly more efficient than *Batch* in resuming the load given that *DR* does not impose any normal operation overhead.

The result for the *Q3* DAG is shown in Figure 25. Again, *Batch*'s resumption time improves in discrete steps based on the input batches that have been processed completely. The performance of *DR* was already explained in Experiment 4 for this DAG. As Figure 25 shows, *Batch* performs better than *DR* for this DAG. The resumption time can also be improved by combining *DR* and *Batch* together (as in *DR-Save*). However, for the *Q3* DAG, the improvement is negligible.

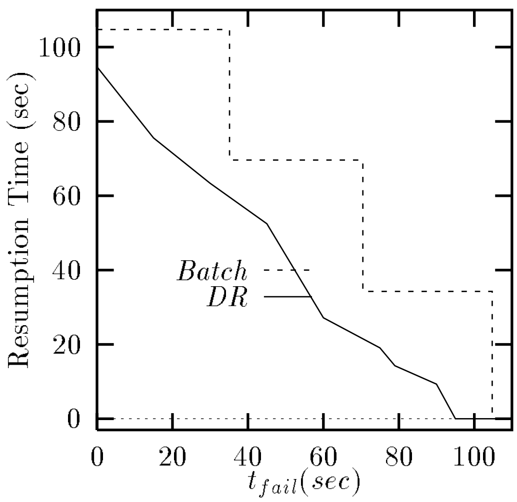

Figure 24: *Batch* vs. *DR* (*Lineitem*)

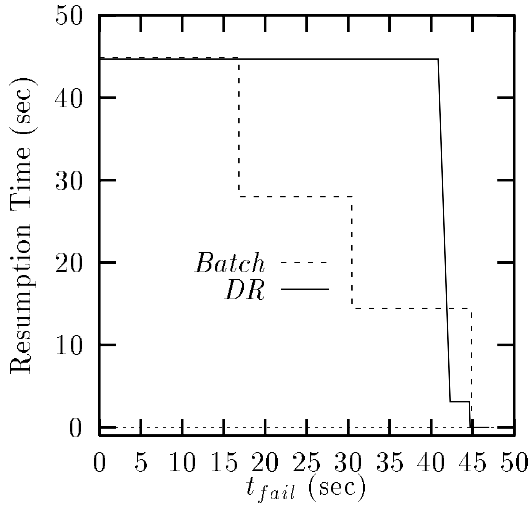

Figure 25: *Batch* vs. *DR* (*Q3*)

Summary: We can draw a number of conclusions from the previous experiments.

- *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.

- 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 and more beneficial. Another reason why a "cost-based" analysis is needed is that in some cases, subset filters may not remove enough tuples to justify the cost that the subset filters impose when a load is resumed (e.g., cost of performing an anti-semijoin).
- In many cases, savepoints (or snapshots) result in a significant normal operation overhead. When a batching algorithm is used, a careful selection of the number of input batches is required because a batching algorithm can result in a significant normal operation overhead. However, if certain transforms of a component DAG are very selective (i.e., few output tuples compared to input tuples), the overhead of savepoints may be tolerable.
- For component DAGs that load dimension and fact tables, *DR*, despite having no normal operation overhead, resumes the load more efficiently than algorithms that employ savepoints or batching. On the other hand, for component DAGs that do not produce warehouse tuples immediately, using savepoints after very selective transforms may be beneficial. In this case, a hybrid algorithm that combines *DR* and the savepoint-based algorithm can be used. For component DAGs that are simple enough (so that input batches can be formed) but do not produce warehouse tuples immediately, a batching algorithm may be best.

7 Conclusions

We developed a resumption algorithm *DR* that performs most of its actions 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 DAG is designed, no matter how many times the *Resume* portion is called to resume from a 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.

DR can also be used to identify "problem spots" in a component DAG, and suggest modifications to make resumption more efficient. For instance, in our example component DAG, transform *TV* needs to reprocess all of its input because *DR* finds that there are no identifying attributes. Further, *TV*'s output is a single tuple, suggesting that saving the result of *TV* is useful. Thus, we are currently augmenting *DR* to selectively record transform outputs using savepoints. We are also improving our implementation of the various filters.

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 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

[BHM90] P. A. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In

Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 112–122, 1990.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan-Kaufman, Inc., San Mateo, CA, 1997.

[Car97] Felipe Carino. High-performance, parallel warehouse servers and large-scale applications, October 1997. Talk about Teradata given in Stanford Database Seminar.

[CLR92] T. H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1992.

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

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufman, Inc., San Mateo, CA, 1993.

[Inc] Sagent Technologies Inc. Personal correspondence with customers.

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

[LWGMG98] 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/ wilburt/-resume.ps.

[MN92] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 361–370, 1992.

[RZ89] 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, September 1989.

[Sag98] Sagent Technology, Inc., Palo Alto, CA. *Sagent Data Mart Population Guide*, 1998.

[WCK93] A. Witkowski, F. Cariño, and P. Kostamaa. NCR 3700 — The Next-Generation Industrial Database Computer. In *Proceedings of the International Conference on Very Large Data Bases*, pages 230–243, 1993.

[WN95] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proceedings of the International Conference on Very Large Data Bases*, pages 30–41, Zurich, Switzerland, 1995. Morgan-Kaufman, Inc.

Algorithm A.1 *AssignReextraction*
Input: Component DAGs G, G'; extractor E
Side effect: Extractor E in G' is assigned a re-extraction procedure
1. If Prefix-feasible(Y_E) and Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetExcl
    2. Assign GetSuffix(Last($\mathcal{C}$),IdAttrs(Y_E)) to E in G'
3. Else If Prefix-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetSuffix
    4. Assign GetDirtySuffix(Last($\mathcal{C}$), IdAttrs(Y_E)) to E in G'
5. Else if Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetSubset
    6. Assign GetSubset($\mathcal{C}$, IdAttrs(Y_E)) to E in G'
7. Else if Prefix-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetSubset
    8. Assign GetDirtySubset($\mathcal{C}$, IdAttrs(Y_E)) to E in G'
9. Else if E supports GetAll
    10. Assign GetAll() to E in G'
11. Else Assign GetAllReordered() to E in G'

Figure 26: Assigning Re-extraction Procedures

A Details of the *DR* Algorithm

In this appendix, we present additional details of the *DR* algorithm.

A.1 Assign Re-extraction Procedures

The full listing of the *AssignReextraction* algorithm is given in Figure 26. Lines 1–2 try to push a clean-prefix filter to the extractor using GetSuffix. Lines 3–4 try to push a dirty-prefix filter to the extractor using GetDSuffix. Lines 5–6 try to push a clean-subset filter to the extractor using GetSubset. Lines 7–8 try to push a dirty-subset filter to the extractor using GetSubset. If no filter can be pushed, *AssignReextraction* tries to assign GetAll. Otherwise, GetAllReordered, which is assumed to be supported, is assigned.

A.2 Removing Redundant Filters

Recall that any filter Y_X^f assigned to Y_X can be one of four filter types: CP, DP, CS, and DS. Since the re-extraction procedures perform the same processing as the input parameter filters, we say that GetSuffix is of filter type CP, GetDirtySuffix is of filter type DP, GetSubset is of filter type CS, and GetDirtySubset is of filter type DS. The GetAll and GetAllReordered re-extraction procedures do not filter any tuples and have no filter types.

The key in removing a redundant filter for Y_X is deducing the filters that are already "in effect" for Y_X due to previous filters or re-extraction procedures. For instance, if a CP filter is assigned to X_V, then Y_X will only receive a suffix of its normal operation input. Thus, even if there is no filter assigned to Y_X, a CP filter is "in effect". If a CP filter is already "in effect", any Y_X^f filter would be redundant since CP filters discard the most tuples. Similarly, if X is an extractor that is assigned GetSuffix, a CP filter is already in effect for Y_X, and any Y_X^f filter would be redundant.

To capture the filters that are in effect, we introduce a field $Y_X.inEffect$ that contains a set of filter types for each input parameter Y_X. (Actually, $inEffect$ also records the attribute sets used by re-extraction procedures and input parameter filters so that redundant filters can be compared appropriately.) Initially, the $inEffect$ field of each input parameter is set to { } by *RemoveRedundantFilters* in Lines 1–2 (Figure 27). The algorithm then computes the filter types in effect due to the re-extraction procedures in Lines 3–5. The algorithm then processes the input

Algorithm A.2 *RemoveRedundantFilters*
Input: Component DAGs G, G'
Side effect: G' with any redundant filters removed

1. For each Y_X in G
    2. $Y_X.inEffect \leftarrow \{\ \}$
3. For each extractor E in G
    4. For each Y_E in G
        Let g be the filter type of E in G'
        Let E use the attribute set A in its re-extraction procedure
        5. $Y_E.inEffect \leftarrow Y_E.inEffect \cup \{\langle g, A\rangle\}$
6. For each Y_X in G in topological order
    7. If Y_X^f is in G' Then
        8. Let g be the filter type of Y_X^f. Let Y_X^f use the attribute set A
        9. If there is a filter type $\langle f, A'\rangle \in Y_X.inEffect$ and $f \succ g$ and $A' \subseteq A$ Then
            10. Remove Y_X^f from G' and connect X to Y in G' // redundant filter removed
        11. Else
            12. $Y_X.inEffect \leftarrow Y_X.inEffect \cup \{\langle g, A\rangle\}$
    13. For each Z_Y in G
        14. $Z_Y.inEffect \leftarrow Z_Y.inEffect \cup Y_X.inEffect$

Figure 27: Removing Redundant Filters

parameters in topological order to ensure that the filter types "in effect" are computed correctly. In Lines 9–10, it checks if the filter Y_X^f is redundant because of previous filters or re-extraction procedures. The effect of previous filters or re-extraction procedures is conveniently recorded in $Y_X.inEffect$. If Y_X^f is redundant, it is removed from G'. Otherwise, Y_X^f stays and the type of filtering it provides is recorded in $Y_X.inEffect$ (Lines 11–14).

A.3 Handling General IdAttrs

Algorithm *DR* only requires changing the implementation of the filters and re-extraction procedures to handle IdAttrs that is a disjunction of attribute sets. The rest of the algorithm, like deciding which filter types and re-extraction procedures to assign, is unchanged.

We now illustrate how the filters are implemented when IdAttrs is a disjunction of attribute sets. Let us suppose that a clean-subset filter Y_X^f is assigned to Y_X. Recall that in the single attribute set case where IdAttrs$(Y_X) = A$, Y_X^f is simply $CS[\mathcal{C}, A]$, and the filter identifies a subset S of the sequence that Y_X receives during resumption time, denoted $\mathcal{Y}'_X$, that can be discarded.

If IdAttrs(Y_X) is a disjunction of attribute sets $A_1 \vee \ldots \vee A_n$, each attribute set A_i identifies a subset S_i of the $\mathcal{Y}'_X$ tuples that can be safely discarded considering one or more paths from Y_X to the warehouse. The problem is that there may be tuples in S_i that cannot be safely filtered when other paths are considered. The solution is to discard only the tuples that can be safely filtered along all paths. That is, only the tuples in $S_1 \cap \ldots \cap S_n$ are filtered.

To implement this solution, each A_i in IdAttrs(Y_X) results in a "sub-filter" denoted $Y_X^i = CS[\mathcal{C}, A_i]$. The overall CS filter Y_X^f then works as follows. For each tuple $x \in \mathcal{Y}'_X$, Y_X^f passes x to each sub-filter Y_X^i. If all sub-filters discard x, then x is discarded. Otherwise, x passes through.

The implementation of other filter types are altered in a similar fashion. For instance, if Y_X is assigned a clean-prefix filter, then the sub-filter Y_X^i is $CP[\text{Last}(\mathcal{C}), A_i]$. The implementation of the re-extraction procedures are also altered in a similar fashion. For more details on how the general case is handled, see [LWGMG98].