Paper Number 149

Efficient Resumption of Interrupted Warehouse Loads*

Wilburt Juan Labio, Janet L. Wiener, Hector Garcia-Molina
Stanford University

Vlad Gorelik
Sagent Technologies

Abstract

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

1 Introduction

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

Warehouse loads are usually performed when the system is off-line (e.g., overnight), and must be completed within a fixed period. A failure during the load creates havoc: Current commercial systems abort the failed load, and the administrator must restart the load from scratch and hope a second failure does not occur. If there is not enough time for the new load, it may be skipped, leaving the database out of date or incomplete, and generating an even bigger load for the next period. Load failures are not unlikely due to the complexity of the warehouse load. For instance, Sagent customers report that one out of every thirty loads fails [11].

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

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

*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.

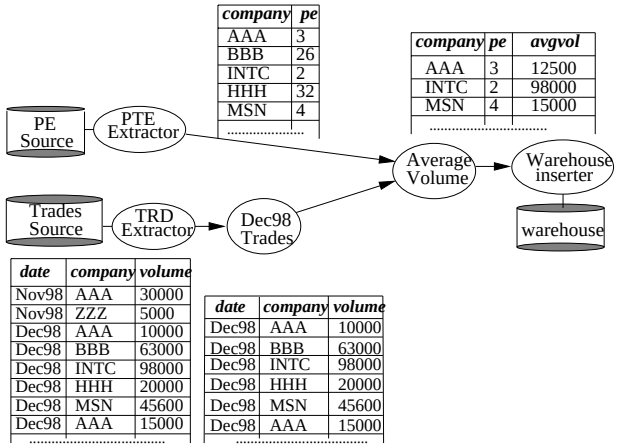

Figure 1: Load Workflow

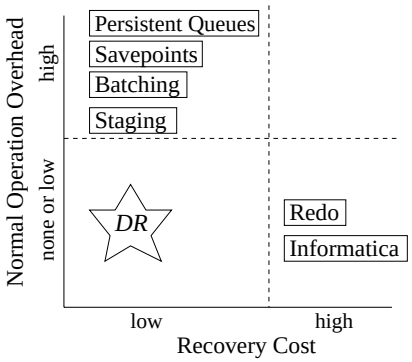

Figure 2: Applicability of Algorithms

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

In practice, load workflows can be much more complex than what we have illustrated, often having tens to hundreds of transforms [11]. Also, the transforms are not just conventional database operations (e.g., join) but are often coded by application specialists to perform arbitrary processing (e.g., data scrubbing, byte reordering). To load the data as fast as possible, the output tuples of each component are sent to the next as soon as they are generated to maximize pipelining. Due to the complexity of the warehouse load and the amount of data loaded, it is not devoid of failures.

There are many ways to recover a failed warehouse load. The fundamental features of various techniques are informally contrasted with our technique, called *DR*, in Figure 2. The vertical axis represents the *normal-operation overhead* of a technique, while the horizontal axis indicates the *recovery cost* of a technique. In the lower right quadrant of Figure 2 are techniques that have very low normal-operation overhead. One such technique is to simply redo the entire load over again. Clearly, this technique can suffer from high recovery cost but it has no normal-operation overhead since it does not modify the load workflow. Informatica's solution [6] 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).

Other techniques, shown in the upper left quadrant of Figure 2, attempt to minimize the recovery cost by aggressively modifying the load workflow or load processing. One such 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. Another technique in the same category is input batching wherein the input to the load workflow is divided into batches, and the batches are processed in sequence. Another technique is to take periodic savepoints [5] of the workflow state, or save tuples in transit in persistent queues [1,2]. When a failure occurs, the

modified transforms cooperate to revert to the latest savepoint, and proceed from there.

In general, techniques that require modification of the load workflow suffer from two disadvantages: (1) the normal-operation overhead is potentially high as confirmed by our experiments; and (2) the specific details of the load processing need to be known. These techniques are not straightforward to implement because careful selection of stages or batches is required to avoid high overhead. Furthermore, since the transforms are not just conventional operations, it is hard to know their specific details.

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

We note that there are previous techniques that are similar to *DR* in that they incur low normal-operation overhead but still have a low recovery cost. However, these techniques are applicable to very specific workflows for disk-based sorting [8], object database loading [12], and loading a flat file into the warehouse [9,13]. Our technique can handle more general workflows.

We do not claim that *DR* always recovers a load faster than other techniques. For instance, since some of the techniques modify the load processing to minimize recovery cost, these techniques can recover a failed load faster than *DR*. As mentioned, the downside of these techniques is the potential high normal-operation overhead and that the load workflow needs to be modified. However, our experiments show that *DR* is competetive if not better than these techniques for many workflows. In particular, *DR* is better for workflows that make heavy use of pipelining. Even if a workflow does not have a natural pipeline, our experiments show that a hybrid algorithm that combines *DR* and staging (or batching) can lower recovery cost.

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

- We develop a framework for describing successful warehouse loads, and load failures. Within this framework, we identify basic properties that are useful in resuming loads.
- We develop *DR* that minimizes the recovery cost while imposing no overhead during normal operation. *DR* does not require knowing the specifics of a transform, but only its basic, high-level properties. *DR* is presented here in the context of warehousing, but is really a generic solution for resuming any long-duration, process intensive task.
- We show experimentally that *DR* can significantly reduce recovery cost, as compared to

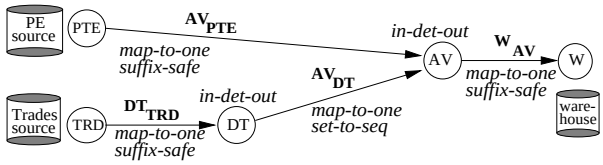

Figure 3: Component Tree with Properties

traditional techniques. In our experiments we use Sagent's warehouse load package to load TPC-D tables and materialized views containing answers to TPC-D queries.

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

2 Normal Operation

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

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

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

2.1 Component Tree Design

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

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

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

As a component tree is designed, the "properties" that hold for each node or edge are declared for use by our resumption algorithm. Commercial load packages already declare basic properties like the key attributes of an input parameter. The properties that *DR* uses are explained in more detail in Section 4. We now illustrate a component tree.

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 $Dec98Trades$), and AV (for $AverageVolume$). The inserter is denoted W.

The input parameter(s) of each component are denoted by the component that produces the input. For instance, AV_{DT} is an input parameter of AV that is produced by DT. 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, and is used as input by AV (i.e., $AV_{DT} = DT_O$).

Figure 3 also shows the properties that hold for each input parameter (edge) and each component (node). For instance, AV_{DT} is map-to-one and set-to-seq. In Section 4, we define these properties and justify why they hold in this example. When the component tree is designed, the attributes and keys of input parameters are also declared. For instance, the attributes of AV_{PTE} tuples are $[company, pe]$, while the key attributes 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 key attributes. W denotes the warehouse inserter.

We note that the component trees designed for warehouse creation and maintenance are different [7]. However, our resumption algorithm applies equally well to both creation and maintenance component trees. 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 tree 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 is 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 component. The next example illustrates a warehouse load during normal operation, i.e., no failures occur.

Example 2.2 Consider the component tree 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 AV_{PTE} is instantiated with the sequence $\mathcal{AV}_{PTE} = \mathcal{PTE}_O$. Note that PTE does not need to produce $\mathcal{PTE}_O$ in its entirety before it can ship a prefix of $\mathcal{PTE}_O$ to AV. Similarly, DT_{TRD} is instantiated with $\mathcal{DT}_{TRD} = \mathcal{TRD}_O$, and so on. Finally, W_{AV} of the inserter is instantiated with $\mathcal{W}_{AV} = \mathcal{AV}_O$. W inserts the tuples in $\mathcal{W}_{AV}$ in order and issues a commit periodically. In the absence of failures, $\mathcal{W}_{AV}$ is eventually stored in the warehouse. □

To summarize our notation, $\mathcal{Y}_X$ and $\mathcal{Y}_O$ denote the sequences used for input parameter Y_X and output parameter Y_O during a warehouse load. When Y produces $\mathcal{Y}_O$ by processing $\mathcal{Y}_X$ (and possibly other input sequences), we say $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$. 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 this paper, we only consider system-level load failures, e.g., RDBMS or software crashes, hardware crashes, lack of disk space. We do not consider load failures due to invalid data. Furthermore, we consider system-level failures that do not affect information stored in stable storage. Any of the components can suffer from such a system-level failure. Even though various components may fail, the effect of any failure on the warehouse is the same. That is, only a prefix of the normal operation input sequence $\mathcal{W}$ is loaded into the warehouse.

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

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

3.1 Data for Resumption

When a component Y fails, the warehouse load eventually halts due to lack of input. Once Y recovers, the load can be resumed. However, very limited data is available to the resumption algorithm. This is because the specific details (e.g., state) of the transforms are not known. 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 procedures are provided in Section 5.3.

- GetAll() 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 GetAll(), that is, that the original data is still available. If $\mathcal{E}_O$ cannot be reproduced, then $\mathcal{E}_O$ must be logged.
- GetAllInorder() extracts the same sequence $\mathcal{E}_O$. This procedure may be supported 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.
- GetSubset(...) provides the $\mathcal{E}_O$ tuples that are not in the subset indicated by GetSubset's parameters. Sources that can selectively filter tuples typically provide GetSubset.
- GetSuffix(...) provides a suffix of $\mathcal{E}_O$ that excludes the prefix indicated by GetSuffix's parameters. Sources that can filter and order tuples typically provide GetSuffix.

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

3.2 Redoing the Warehouse Load

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

Although *Redo* is very simple, it still requires that the entire workflow satisfies the property that if the same set of tuples are obtained by the extractors, the same set of tuples are inserted into the

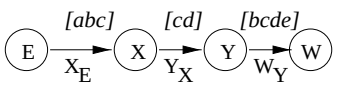

Figure 4: Component Tree

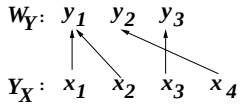

Figure 5: Map-to-one

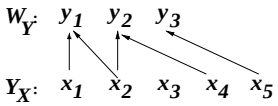

Figure 6: Suffix-safe

warehouse. Since this property pertains to an entire workflow, it can be hard to test. A *singular property* that pertains to a single transform is much easier to test. The following singular property, set-to-set, is sufficient to enable *Redo*. That is, if all extractors use GetAll or GetAllInorder, and all transforms are set-to-set, then *Redo* can be used. This condition is tested in Definition 3.1

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

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

4 Properties for Resumption

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

To illustrate, consider the simple component tree in Figure 4. Suppose that the sequence $\mathcal{W}_Y$ to be inserted into the warehouse is $[y_1y_2y_3]$, and $[x_1x_2x_3x_4]$ is the $\mathcal{Y}_X$ input sequence that yields the warehouse tuples (Figure 5). 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 in the warehouse. Clearly, it is safe to filter $\mathcal{Y}_X$ tuples that contribute only to $\mathcal{W}_Y$ tuples, such as y_1, already in the warehouse. Thus in Figure 5, x_1 and x_2 can be filtered. We need to be careful with y_1 contributors that also contribute to other $\mathcal{W}_Y$ tuples. For example, if x_2 contributes to y_2 as well, then we cannot filter x_2, since it is still needed to generate y_2.

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

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

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

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

In Section 4.1, we identify four singular properties to answer Question (2). We then define three *transitive properties* that apply to sub-trees of the component tree. *DR* will derive the transitive properties based on the declared singular properties. In Section 4.2, we define two more singular properties. Using these properties, we define *identifying attributes* of the tuples to answer Question (1). *DR* will derive the identifying attributes based on the declared singular properties and key attributes. In Section 6, we present a study that shows that the singular properties hold for many commercial transforms. Since singular properties pertain to a transform or an input parameter and not to a whole workflow, they are easy to grasp and can often be deduced easily from the transform manuals. Henceforth, we refer to singular properties as "properties" for conciseness.

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

Definition 4.1 (**Contributes, Contributors**) Given transform Y, let $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$ and $Y(...\mathcal{Y}'_X...) = \mathcal{Y}'_O$. Also let $\mathcal{Y}_X = [x_1..x_{i-1}x_ix_{i+1}..x_n]$ and $\mathcal{Y}'_X = [x_1..x_{i-1}x_{i+1}..x_n]$.

Contributes(x_i,y_j) = true, if $y_j \in \mathcal{Y}_O$ and $y_j \notin \mathcal{Y}'_O$. Otherwise, Contributes(x_i,y_j) = false.

Contributors($\mathcal{Y}_X$, y_j) = $\mathcal{T}$, where IsSubsequence($\mathcal{T}$, $\mathcal{Y}_X$) and ($\forall x_i \in \mathcal{T}$: Contributes(x_i, y_j)) and ($\forall x_i \in \mathcal{Y}_X$: Contributes(x_i,y_j) $\Rightarrow$ $x_i \in \mathcal{T}$). □

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

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

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.

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

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

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

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 $[y_1 y_2]$ does not need to be produced, the prefix $[x_1 x_2 x_3 x_4]$ 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 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 $[x_5 x_4 x_3 x_2 x_1]$ 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. Many transforms satisfy this property (see Section 6).

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, the same output sequence is produced by a sorting transform as long it processes the same set of input tuples.

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

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

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

4.2 Identifying Contributors

To determine which $\mathcal{Y}_X$ tuples contribute to a warehouse tuple w_k, we are only provided with the value of w_k after the failure. Since transforms are black boxes, the only way to identify the contributors to w_k is to match the attributes that the $\mathcal{Y}_X$ tuples and w_k have in common. (If a transform changes an attribute value, e.g., reorders 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) ∩ Attrs(Y_O). Selection, projection, aggregation, and union transforms have input parameters with no hidden contributors (see Section 6).

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) ∩ 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. We now illustrate how the potential contributors are found.

Example 4.1 Consider again the simple component tree shown in Figure 4. The labels above the edges give the attributes of the input tuples, e.g., Attrs(Y_X) = [cd]. If Y_X has no hidden contributors, then all of the $\mathcal{Y}_X$ contributors to a warehouse tuple w_k, denoted S_k, match w_k on [cd] (i.e., Attrs(Y_X) ∩ Attrs(Y_O)). If X_E has no hidden contributors, then all of the $\mathcal{X}_E$ contributors to $x_i \in S_k$ match x_i on [c] (i.e., Attrs(X_E) ∩ Attrs(X_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 $\mathcal{X}_E$ tuples that contribute to w_k match w_k on [c]. Hence, the potential contributors of w_k in $\mathcal{X}_E$ are exactly the ones that match w_k on [c]. □

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

Definition 4.5 (**CandAttrs(Y_X)**) There are three possibilities for CandAttrs(Y_X):

1. If Y is the warehouse inserter, then CandAttrs(Y_X) = Attrs(Y_X).
2. If Y_X has hidden contributors then CandAttrs(Y_X) = [ ].
3. Else CandAttrs(Y_X) = CandAttrs(Z_Y) $\cap$ Attrs(Y_X). □

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

Since the potential contributors identified by CandAttrs(Y_X) 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 may also be used 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 again the component tree shown in Figure 4. Note that CandAttrs(X_E) = [c] assuming that X_E, Y_X, and W_Y have no hidden contributors. Now consider which attributes can be used as IdAttrs(X_E). There are three possibilities.

1. IdAttrs(X_E) = KeyAttrs(X_E) if KeyAttrs(X_E) $\subseteq$ CandAttrs(X_E) and both X and Y satisfy the no-spurious-output property.
2. IdAttrs(X_E) = KeyAttrs(W_Y) if KeyAttrs(W_Y) $\subseteq$ CandAttrs(X_E).
3. IdAttrs(X_E) = IdAttrs(Y_X) if IdAttrs(Y_X) $\subseteq$ CandAttrs(X_E).

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

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

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

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

The example above provides the intuition behind our definition of the identifying attributes of Y_X.

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

1. If (KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X) and $\forall Z_V \in P : Z_V$ has no spurious output tuples), then IdAttrs(Y_X) = KeyAttrs(Y_X).
2. Otherwise, let $Z_V \in P$ but $Z_V \neq Y_X$.
   If IdAttrs(Z_V) $\neq$ [ ] and IdAttrs(Z_V) $\subseteq$ CandAttrs(Y_X), then IdAttrs(Y_X) = IdAttrs(Z_V).
3. Otherwise IdAttrs(Y_X) = [ ]. □

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

4.3 The Trades Example Revisited

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

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

Table 1: Properties and Functions of Transforms.

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

Table 2: Declared and Inferred Properties of Input Parameters.

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 both 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 tree 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 1998. 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 AV reads each tuple in AV_{PTE} and if its pe attribute is ≤ 4, it finds all of the trade tuples for the same company in AV_{DT}, which are probably not in order by company. AV then groups the AV_{DT} tuples and computes the average trade volume for each company. Then it processes the next tuple in AV_{PTE}. AV_{PTE} is map-to-one since each tuple contributes to zero or one output tuple.. It is not set-to-seq for the same reason it is suffix-safe: AV processes tuples from AV_{PTE} one at a time, in order. AV_{DT} is map-to-one since each trade tuple contributes to the average volume tuple of only one company. However, AV_{DT} is not suffix-safe, e.g., the trade tuple needed to join with the first tuple in AV_{PTE} may be the last tuple in AV_{DT}. Similarly, it is set-to-seq because the order of trades tuples is not relevant to AV. Finally, since the warehouse inserter simply stores its input tuples in order, W_{AV} is map-to-one and suffix-safe but not set-to-seq.

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

5 The DR Resumption Algorithm

We now present the DR resumption algorithm, which uses the properties developed in the previous section. DR is actually composed of two algorithms, *Design* and *Resume*, hence the name. After a component tree G is designed, *Design* constructs a component tree G' that *Resume* will employ to resume any failed warehouse load that used G. The component tree G' is the same as G except for the following differences: (1) Re-extraction procedures are assigned to the extractors in G'; and (2) Filters are assigned to some of the input parameters in G'.

The component tree 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. After this illustration, we cover DR in more detail. *Design* 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 7.

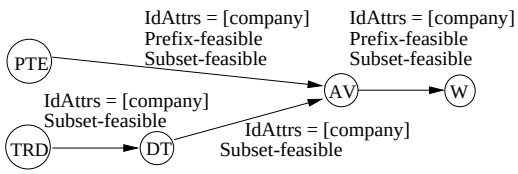

Figure 7: IdAttrs and Transitive Properties

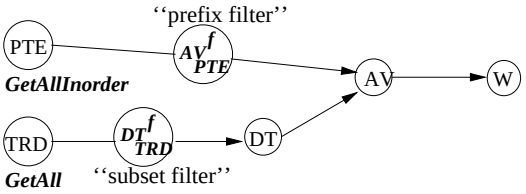

Figure 8: Procedures and Filters Assigned

Design then constructs G' by first assigning re-extraction procedures to extractors based on the computed properties and identifying attributes. Since IdAttrs(AV_{PTE}) = [*company*], it is possible to identify source PE tuples that contribute to tuples in the warehouse based on the *company* attribute. Since both Prefix-feasible(AV_{PTE}) and Subset-feasible(AV_{PTE}) hold, *Design* can assign either GetSuffix or GetSubset to PTE to avoid re-extracting all the PE tuples. However, suppose PTE supports neither GetSuffix nor GetSubset. GetAllInorder is assigned to PTE instead.

Since Subset-feasible(DT_{TRD}) holds and IdAttrs(DT_{TRD}) = [*company*], GetSubset can be assigned to TRD. GetSuffix cannot be assigned since Prefix-feasible(DT_{TRD}) does not hold. Assume TRD does not support GetSubset nor GetAllInorder. GetAll is 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 GetAll, so it is not possible to filter a prefix of the DT_{TRD} input sequence. Since DT_{TRD} is Subset-feasible and IdAttrs(DT_{TRD}) = [*company*], it is possible to filter a subset of the DT_{TRD} input sequence by using *company* to identify the contributors to the warehouse tuples. Thus, a filter, denoted DT^f_{TRD}, that removes a subset of the DT_{TRD} input sequence is assigned to DT_{TRD}. When a failed load is resumed, DT^f_{TRD} removes all tuples in the DT_{TRD} sequence whose *company* attribute value matches some warehouse tuple.

2. Same-seq(AV_{PTE}) holds because PTE is assigned GetAllInorder. Therefore, AV_{PTE} is both Prefix-feasible and Same-seq, so it is possible to filter a prefix of the AV_{PTE} input sequence. Furthermore, we can identify the contributors based on IdAttrs(AV_{PTE}) = [*company*]. Therefore, a filter AV^f_{PTE} that removes a prefix of the AV_{PTE} input sequence, is assigned to AV_{PTE}. When a failed load is resumed, AV^f_{PTE} removes the prefix of the AV_{PTE} input sequence that ends with the tuple whose *company* attribute matches the last warehouse tuple.

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

4. A subset filter can also be assigned to W_{AV}. However, *Design* determines that this filter is redundant with the previous filters. Therefore, no filter is assigned to W_{AV}.

The component tree G' constructed from G is shown in Figure 8. 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' is negligible compared

to the time to design and debug G, which is in the order of weeks or months [11]. 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

$$\mathcal{C} \;=\; [\,\langle AAA, 3, 12500\rangle\ \langle INTC, 2, 98000\rangle\ \langle MSN, 4, 15000\rangle\,],$$

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

Subset filter DT^f_{TRD} is instantiated to remove any DT_{TRD} 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(DT_{TRD})), and that tuple is in $\mathcal{C}$. Similarly, prefix filter AV^f_{PTE} is instantiated to remove the prefix of its AV_{PTE} input that ends with the tuple whose *company* attribute is MSN. It is safe to filter these tuples since it is guaranteed that AV has processed through the MSN tuple of AV_{PTE} (i.e., Prefix-feasible(AV_{PTE})). Note that the tuples before the MSN tuple may include ones that do not contribute to any warehouse tuple (i.e., 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 DT, AV and W. Had the load failed with a longer warehouse tuple sequence $\mathcal{C}$, the filters would have been instantiated appropriately by *Resume* to filter more input tuples.

In summary, DR avoids re-processing many of the input tuples using filters. Also, had the extractors PTE and TRD supported GetSubset or GetSuffix, DR could have even avoided re-extracting tuples from the sources. Furthermore, DR does not incur normal operation overhead, nor rely on identifying appropriate savepoints or input batches.

5.2 Filters

In the example, we mentioned subset filters and prefix filters. There are two types of subset filters and two types of prefix filters that may be assigned to Y_X. In each case, the filter 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[Last($\mathcal{C}$), 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[Last($\mathcal{C}$), 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 trees 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 9: 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 but not Same-seq 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 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 are as follows: $CP \succ DP \succ DS$, $CP \succ CS \succ DS$.

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

The procedure *AssignFilter* is shown in Figure 9. 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.

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

Algorithm 5.2 *Design*
Input: Component tree G
Output: Component tree 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 tree 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 10: *DR* Algorithm

correspond to the DP and DS filters.

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

□

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* try 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 GetAllInorder or GetAll is assigned to it. The detailed listing of *AssignReextraction* is in [7].

5.4 The *Design* and *Resume* Algorithms

Algorithm *Design* of *DR* (Algorithm 5.2, Figure 10) constructs G' by processing the component tree G in two passes. In reverse topological order, *Design* computes IdAttrs, Prefix-feasible and Subset-feasible (Line 2). It then uses *AssignReextraction* to assign the procedures to the extractors (Lines 3–4). In topological order, it computes Same-seq and uses *AssignFilters* to assign filters to input parameters (Lines 5–8). In case of failure, *Resume* of *DR* (Algorithm 5.3, Figure 10) 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.

We now discuss how redundant filters are detected by *Design* (Line 9). 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:

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. In [7], we show how redundant filters can be removed in $O(n^3)$ time, where n is the number of nodes in G.

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

6 Experiments

In this section, we present our experiments which compare *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 trees 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 GetAllInorder and GetAll, but only the "SQL" Extractor supports GetSuffix, GetDirtySuffix, GetSubset and GetDirtySubset. The 19 transforms include Sagent's implementations of conventional operations used in databases, such as selection, projection, union, aggregation, and join. The 19 transforms have a total of 23 input parameters. A summary of the properties that hold for the transforms and the input parameters are listed in Figure 11.

- 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 11: Properties of Sagent Transforms and Input Parameters

Some of the properties, like suffix-safe, are actually declared by Sagent. Other properties were deduced easily from the Sagent manuals that specify the transforms. The statistics above 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 scenarios.

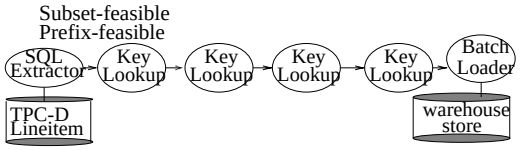

Figure 12: Fact Table Creation

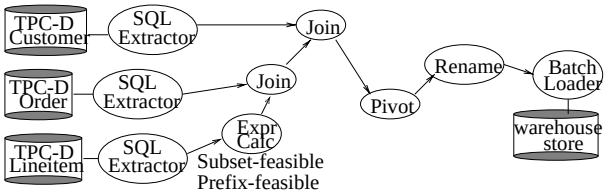

Figure 13: TPC-D View Creation

6.2 Resumption Time Comparison

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

Overview: We designed three types of component trees. One type of component tree loads *dimension tables*, e.g., the *Customer* and *Supplier* TPC-D tables [4]. Dimension tables typically store data about entities like customers. Another type of component tree loads *fact tables*, e.g., the *Order* and *Lineitem* TPC-D tables. Fact tables typically store transactional data. The last type of component tree 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 trees for loading the TPC-D fact table *Lineitem*, and the materialized view for the TPC-D query *Q3* are shown in Figures 12 and 13 respectively. Query *Q3*, the "shipping priority query," joins 3 tables and performs a GROUP BY and a SUM of revenue estimates.

We performed three sets of experiments. In the first set, we compared *DR* to the algorithms that impose no normal operation overhead (i.e., the lower right quadrant of Figure 2, Section 1). We then compared *DR* to "staging" (with savepoints) which is an algorithm in the upper left quadrant of Figure 2. Finally, we compared *DR* to "batching" which is another algorithm in the upper left quadrant of Figure 2.

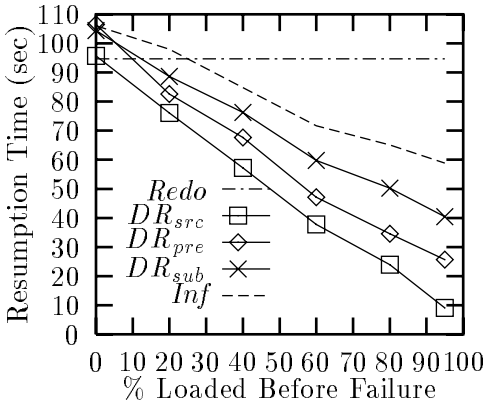

Figure 14: Resumption Time (*Lineitem*)

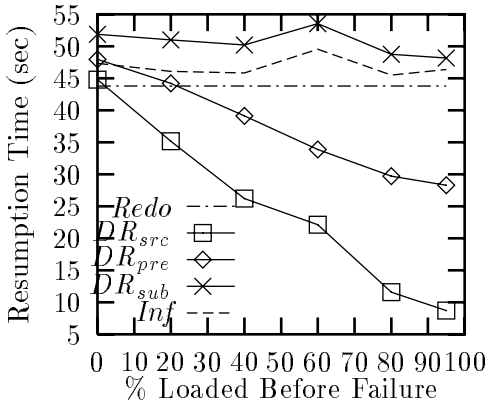

Figure 15: Resumption Time (*Q3*)

Experiment 1: In the first experiment, we compared the resumption times of *DR*, *Redo*, and *Inf* for the *Lineitem* tree (Figure 12). *Inf* is the algorithm used by Informatica [6], and uses only

one filter just before the inserter. We also studied "variants" of *DR* to evaluate the importance of source and prefix filtering. Variant DR_{src} pushes filtering to the *Lineitem* extractor. DR_{pre} uses a prefix filter right after the *Lineitem* extractor, while DR_{sub} uses a subset filter. Normally, given the properties derived for the *Lineitem* tree, *DR* would employ the filtering done by DR_{src}. 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 14, 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* performs 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 faster than *Redo*, DR_{pre} resumes the load 3.68 times faster, and DR_{sub} resumes the load 2.35 times faster. *DR* also resumes 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. Still, DR_{src} is almost as fast as *Redo* when the warehouse table is 0% 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 expensive anti-semijoin operation employed by DR_{sub}'s filters.

Experiment 2: The second experiment is similar to the first but considers the *Q3* tree (Figure 13). The results are shown in Figure 15. 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. 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} and *Inf* do not remove enough tuples to compensate for the cost of the filters.

Experiment 3: In the third experiment, we examined the normal operation overhead of a recovery algorithm that is based on savepoints. We again considered the *Lineitem* and *Q3* component trees. For the former component tree, 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 16. 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* tree, 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 tree. Even with three savepoints, the normal operation load is only about 1.08 times slower (see Figure 17). 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 16: 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 17: 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* tree, 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* tree, 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* trees. In the case of *Save*, we used two savepoints for each component tree.

The result for the *Lineitem* tree is shown in Figure 18 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* tree, *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* tree is shown in Figure 19. Again, *Save*'s resumption time improves in discrete steps based on the completion of the savepoints. For this tree, *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 tree, *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* tree into three "sub-trees." 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-tree" that produced it as if it was a warehouse table being loaded by a component tree. The performance of the hybrid algorithm, denoted *DR-Save*, is plotted in Figure 19. For most values of t_{fail}, *DR-Save* is better than either *Save* or *DR*.

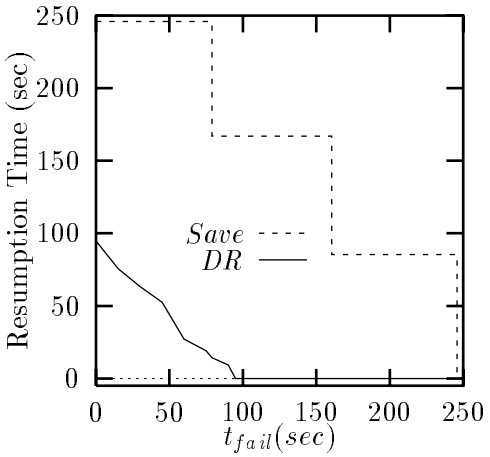

Figure 18: *Save* vs. *DR* (*Lineitem*)

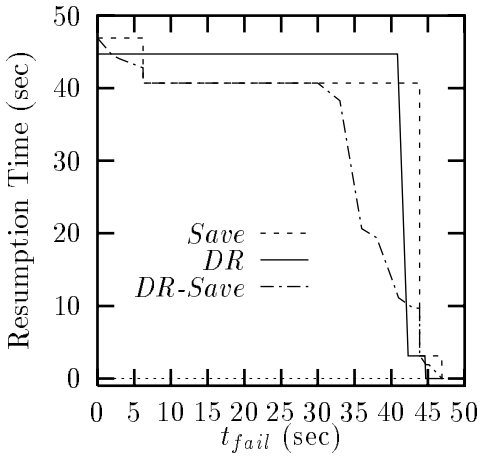

Figure 19: *Save* vs. *DR* (*Q3*)

Experiment 5: In the fifth experiment, we examined the normal operation overhead of a recovery algorithm that is based on batching. We again considered the *Lineitem* and *Q3* component trees. For the former component tree, we loaded *Lineitem* in three input batches. The results are shown in Figure 20. The table shows that batching results in a significant overhead especially when 4 or more batches are used.

For the *Q3* tree, we also loaded the target table using three input batches. The results are shown in Figure 21. 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

# 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 20: 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 21: Batching Overhead (*Q3*)

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 tree 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

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

The result for the *Lineitem* tree is shown in Figure 22 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* tree, *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* tree is shown in Figure 23. 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 tree. As Figure 23 shows, *Batch* performs better than *DR* for this tree. The resumption time can also be improved by combining *DR* and *Batch* together (as in *DR-Save*). However, for the *Q3* tree, the improvement is negligible.

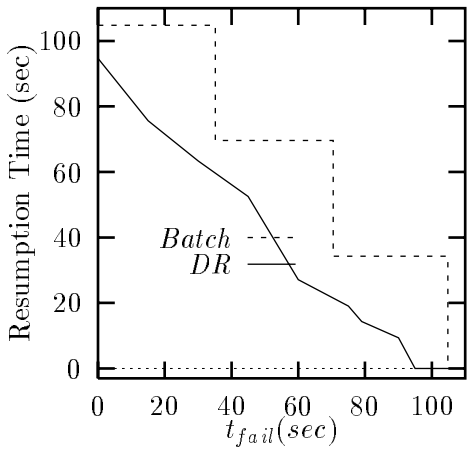

Figure 22: *Batch* vs. *DR* (*Lineitem*)

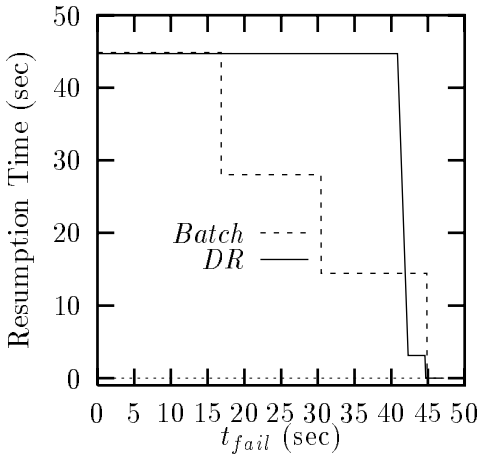

Figure 23: *Batch* vs. *DR* (*Q3*)

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 tree are very selective (i.e., few output tuples compared to input tuples), the overhead of savepoints may be tolerable.

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

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

Finally, our experiments showed that a hybrid algorithm that combines *DR* with staging and/or savepoints maybe beneficial. We are currently augmenting *DR* to selectively record transform outputs using savepoints. We are also improving our implementation of the various filters.

References

[1] 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.

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

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

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

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

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

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

[8] 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.

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

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

[11] S. Technologies. Personal correspondence with customers.

[12] 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.

[13] 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.