PROQID: Partial Restarts of Queries in Distributed Databases

Jon Olav Hauglid
Dept.of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
joh@idi.ntnu.no

Kjetil Nørvåg
Dept.of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
noervaag@idi.ntnu.no

ABSTRACT

In a number of application areas, distributed database systems can be used to provide persistent storage of data while providing efficient access for both local and remote data. With an increasing number of sites (computers) involved in a query, the probability of failure at query time increases. Recovery has previously only focused on database updates while query failures have been handled by complete restart of the query. This technique is not always applicable in the context of large queries and queries with deadlines. In this paper we present an approach for *partial restart* of queries that incurs minimal extra network traffic during query recovery. Based on results from experiments on an implementation of the partial restart technique in a distributed database system, we demonstrate its applicability and significant reduction of query cost in the presence of failures.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Reliability, Performance

Keywords

Distributed querying, fault-tolerance, query restart

1. INTRODUCTION

In a number of application areas, distributed database systems can be used to provide a combination of persistent data storage and efficient access to both local and remote data. Traditionally, both updates and queries have been characterized by short transactions accessing only small amounts of data. This has changed with the emergence of application areas such as Grid databases, distributed data warehouses and peer-to-peer databases. In these areas, time-consuming queries involving very large amounts of data can be expected. With an increasing number of sites (computers) involved in a query, the probability of failure increases. The probability also increases with longer duration of queries and/or higher churn rate (unavailable sites).

Previously, failure of queries has been handled by complete query restart. While this is an appropriate technique for small and medium-sized queries, it can be expensive for very large queries, and in some application areas there can also be deadlines on results so that complete restart should be avoided. An alternative to complete restart is a technique supporting *partial restart*. In this case, queries can be resumed on new sites after failures, utilizing partial results already produced before the failure. These results can be both results generated at non-failing sites as well as results from failing sites that have already been communicated to non-failing sites.

In this paper we present an approach to partial restart of queries in distributed database systems (PROQID) which is based on deterministic tuple delivery and caching of partial query results. The proposed approach has been implemented in a distributed database system, and the applicability of the approach is demonstrated from the experimental results which shows that query cost in the presence of failures can be significantly reduced. The main contribution of our work is a technique for partial restart that: 1) Reduces query execution time compared to complete restart, 2) Incurs only marginal extra network traffic during recovery from query failure, 3) employs decentralized failure detection, 4) supports non-blocking operators and 5) handles recovery from multi-site failures.

While we in this paper focus on using our technique for reducing query cost in the context of failure during execution of large queries, we also note that the technique can be applied to solve a related problem: distributed query suspend and resume, where ongoing low-priority queries can be suspended when higher-priority queries arrive. In this case our approach can be used to efficiently store the current state of the query with subsequent restart from the current state.

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. In Section 3 we outline the assumed system model and query execution. In Section 4 we discuss how failures can be detected, which sites can fail and different times of failure. In Section 5 we describe how to support partial restart. In Section 6 we describe how to handle partial restart in the context of updates during the query processing. In Section 7 we evaluate the usefulness of our approach. Finally, in Section 8, we conclude the paper and outline issues for further work.

2. RELATED WORK

Previously in database systems, only fault tolerance with respect to updates has been considered. This has been solved through the concepts of atomic transactions, usually supported by logging of operations which can be used in the recovery process. Recently, in particular in the context of internet applications, persistent ap-

CIKM'08, October 26–30, 2008, Napa Valley, California, USA.

plication and application recovery has received some attention [2, 18].

Much of the previous work on distributed database systems is obviously relevant. For a survey of state of the art in this area we refer to [13]. Recent work in this area includes query processors for Internet data sources, for example ObjectGlobe [4], and query execution in Grid databases [9].

A related work that directly addresses the issues regarding query restart, is the work of Smith and Watson on fault-tolerance in distributed query processing [17]. Their work is based on GSA-DQP [1], a distributed query processing system for the Grid. Smith and Watson describe how they have extended this system to support fault-tolerance during query execution. Each site executing a query operator, stores tuples produced in a recovery log before sending them towards the query initiator. When a failure is detected by the global fault detector, a new site is selected to replace the failed site. Recovery logs from upstream sites are then replayed so that the new site can restart the failed operation. This assumes that the operations produce tuples in deterministic order, however, how this can be achieved is not described. In contrast to our approach, the new site will produce duplicate tuples in cases where the failed operation was partially completed. These are afterwards discarded at downstream sites by using a system of checkpoint marker identifiers. Our approach improves on the approach of Smith and Watson by avoiding duplication of tuples, employing decentralized failure-detection, supporting pipelined hash-join, and handling multi-site failures.

A similar approach is presented by Hwang et. al. [10, 11]. Their focus is on recovery from failures in a distributed stream processing setting. Three alternatives are presented, two which focus on having standby sites for all sites participating in the query. The third, *upstream backup*, is related to our tuple cache. In contrast to our work, their focus is on fast recovery at the expense of increased network load during execution and failure handling.

While our approach relies on the DBMS to make use of replication in order to handle failures, an alternative is returning partial results [3]. The authors present an approach where the partial result can be resubmitted by the application at a later time in order to hopefully access previously unavailable data sources. The presented solution does not handle failures during query processing, a data source is assumed to either be available or unavailable for the entire query execution.

Related to query restart because of failures, is *suspend and resume*, typically done because of the arrival of a higher-priority query. In [5] the problem is discussed in context of a centralized system, where variants of lightweight checkpointing are used to support resume. A similar centralized approach is described in [6]. Here restart cost is minimized by saving a bounded number of tuples produced by filter operators (intermediate or final results). These tuples are chosen so that they maximize the number of source tuples that can be skipped during restart of scan operations. In contrast to our work, these approaches deal with restarts caused by planned suspend rather than site failures. Also, they don't look at restart of complete queries where our methods can also handle restart of (unfinished) subqueries.

Several approaches to restart loading after failure during loading of data warehouses have been presented, for example [14]. In contrast to our work, the same source sites are restarted after failure, only failing sources sites are considered, and which source tuples can be skipped have to be computed after failure.

Using stored results are also done in a number of other contexts, for example in semantic caching [8, 15], use of materialized views in queries [7], query caching [16], and techniques particularly useful for large and long-running queries, for example reoptimization [12].

Traditionally, also various checkpoint-restart techniques have been employed to avoid complete restart of operations. However, these techniques have been geared towards update/load operations, and assuming the checkpointing information is stored locally, the query will be delayed until the failed site is online again.

Symbol	Description
S_i	Site
t_i	Tuple
T	Table T
T_i	Fragment i of table T
$T_{i(j)}$	Replica j of fragment i of table T
n_i	Algebra node
N_i	Algebra tree

Table 1: Symbols.

3. PRELIMINARIES

In this section we outline the system model that provides the context for the rest of the paper and basic assumptions regarding query execution. We also introduce symbols to be used throughout the paper, summarized in Table 1.

3.1 System Model

The system consists of a number of *sites*, S_i. Each site has a DBMS, and a site can access local data and take part in the execution of distributed queries, i.e., the DBMSes together constitute a distributed database system. The distribution aspects can be supported directly by the local DBMS or can be provided through middleware.

The degree of autonomy is orthogonal to the techniques presented in this paper, i.e., the sites can be completely autonymous or managed by a single entity. The only requirement is that they have a common protocol for execution of queries and metadata management.

Our approach assumes that data can be represented in the relational data model, i.e., tuples t_i being part of a table T. A table can be stored in its entirety on one site, or it can be horizontally fragmented over a number of sites. Fragment i of table T is denoted T_i. Vertical fragmentation can be considered as redesign of tables and require no special treatment in our context.

In order to improve both performance as well as availability, fragments can be replicated, i.e., a part of a table can be stored in more than one site. Replica j of fragment T_i is denoted $T_{i(j)}$.

3.2 Query Execution

We assume queries are written in some language that can be transformed into relational algebra operators, for example SQL. During query planning the different *algebra nodes* n_i are assigned to sites. This requires catalog lookups in order to transform logical table accesses into physical localization programs (for example a set of accesses to horizontal table fragments). We assume that sites can be assigned more than one algebra node so that one site can be assigned a subquery. As all sites have the capability to execute operators, sites containing table fragments used in the query are typically also assigned query operations on these fragments during planning. This tends to reduce network traffic as tuples can be processed locally. Detailed algorithms for assigning nodes to sites are beyond the scope of this paper. Example algebra with site assignment is shown in Figure 1.

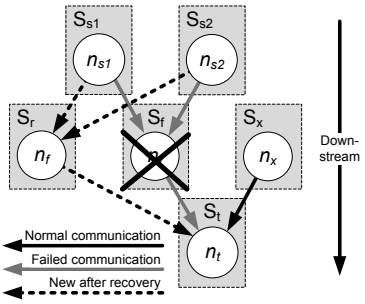

Figure 1: Algebra example.

After planning, query execution begins by transmitting the algebra tree N_i from the initiator site to the different sites involved, as will be described in more detail in the next section.

Results of query operators are transferred between sites in *tuple packets*. These packets are atomic, i.e., incomplete packets are discarded. Our approach supports stream-based processing of tuples, for example joins performed by pipelined hash-join [19]. This means that an algebra node can start producing tuples before the all tuples are available from its operand node(s). This makes it possible for nodes downstream to start processing as soon as possible and therefore lets more nodes execute in parallel. This requires each site to be able to accept and buffer yet unprocessed packets, but it allows data transfers to be made without explicit requests, thereby improving response time. In case of limited buffer availability, flow control can be used to temporarily halt sending of packets.

4. FAILURES AND FAILURE DETECTION

Obviously, a precondition for failure recovery is failure detection. In this section we describe how failures can be detected, which sites can fail and different times of failure.

In our setting, a site will be considered failed if it cannot be contacted by other sites, i.e., failure can be both network and actual site failure. Observed from the outside the result is the same.

We propose an approach to decentralized failure detection and handling where each site is responsible for detecting failures in sites that are supposed to produce its input tuples. The site responsible for discovering failure of a site S_i is called the *failure detector site* of S_i. The result is that failures in two separate branches of the algebra tree are detected and handled separately. It also removes a global failure detector as a single point of failure. Note that the status of sites not participating in any query does not have to be maintained.

Since a failure detector site expects tuples from sites it is watching, detection can be implemented as timeouts on these tuples as this incurs no extra overhead. Note that blocking operations (e.g. aggregation), can result in timeouts even if the site is alive. Therefore a timeout is followed by a special alive check message to verify that a site (or connection) has indeed failed. Timeout on the response of this message constitutes a suspected failure and failure handling should be started.

Decentralized failure handling depends on each site being able to replan a (sub-)query after it detects failures. In order to do this each algebra node must be aware of its algebra subtree(s). This can be achieved by a stepwise transmission of the algebra tree from the initiator site to the different sites involved in a query. This process is described in Algorithm 1. A site assigned a given algebra node transmits this node's subtree(s) to the site(s) assigned the root(s) of these, all while maintaining a local copy of the subtree(s). The local copies are then used during failure handling, as described in the next section.

Algorithm 1 Stepwise transmission of algebra tree.

At site S_i, after receiving N_i:

$n_i \leftarrow root(N_i)$
for all $n_c \in children(n_i)$ **do**
  $N_c \leftarrow subtree(n_c)$
  $S_c \leftarrow getAssignedSite(n_c)$
  $Send(N_c, S_c)$
end for

The initiator site starts by executing the algorithm with the whole algebra tree as N_i.

For a given query, involved sites can fail either during transmission of the algebra tree or during query execution. During execution, there are essentially three different types of sites that can fail: sites containing 1) the initiator node, 2) table fragment access nodes, and 3) operator nodes.

4.1 Failure During Transmission of Algebra Tree

A site can fail before it receives its algebra node(s). This will be detected by its failure detector site during the stepwise transmission of the algebra tree. The detector site starts failure handling by replanning the failed subtree and transmits it to the replacement site.

Note that the stepwise transmission of the algebra tree makes handling of this kind of failure easier compared to a parallel transmission of all algebra nodes directly from the initiator site. This is because failure detection and handling is decentralized and that no sites upstream from a failed site will receive their algebra nodes before the failed site has been replaced and the query plan updated.

4.2 Failure of Site Containing Initiator Node

If the site with the initiator node fails, the query should simply abort as the result is now of no interest. Initiator failure is detected when sites with algebra nodes directly upstream from the initiator site try to send their results. This is the only failure which is detected by a site upstream rather than downstream. After detection, an abort message is sent upstream to the other sites participating in the query.

4.3 Failure of Site Containing Table Fragment Access Node

A site assigned a table fragment access node could fail. When this is detected by the site with the algebra node that should get the results, the detector site should find a different site with a replica of the same fragment and send the table fragment access operator to the new site for restart.

4.4 Failure of Site Containing Operator Node

A site responsible for an operator node n_f can fail. In these kinds of failures, four types of sites are involved (Cf. Table 2 and Figure 1).

When failure of an operator is detected, the detector site must find a replacement site and send it the relevant algebra subtree. This is made possible by each site storing the subtree(s) for its assigned algebra node as described above. The replacement site must inform sites upstream about its existence and do a complete or partial restart of the operation depending on time of failure. This process is described in the next section.

Symbol	Description
S_f	Failed site, assigned the algebra node n_f.
S_s	Site(s) with the source algebra node(s) n_s (i.e., producing tuples for n_f).
S_r	Site replacing S_f. Will restart n_f.
S_t	Site with the algebra node receiving the results from n_f. Is failure detector for S_f.

Table 2: Sites involved in a failure.

5. RESTART AFTER FAILURE

By supporting partial restart of queries, one site is able to pick up and continue an operation after the site that originally executed the operation has failed. We now give a brief overview on how to perform query restart before we delve into the details in the following sections.

For now we assume that each site only has a single algebra node. This node, as described in Section 4, can be either a local table fragment access or an operator node. This restriction is lifted in Section 5.5 where we look at handling multiple failures.

With local table fragment access, restarting is independent of other sites. After S_s detects a failure of S_f, it must locate a suitable S_r using catalog lookup and transmit the table fragment access operation to it so that the operation can be restarted.

In the case of operator failure, the failed node had one or more operands located on source site(s) S_s. As part of the restart, it is necessary to inform these of the new S_r so that operand tuples can be sent there. The amount of tuples that must be sent from S_s depends on the failed operation and will be discussed in Section 5.2.

In the worst case, all tuples that S_s ever sent to S_f must be resent to S_r. If no measures were taken to prevent it, S_s would then be required to completely restart its/their operation(s). If these operations were not local table fragment accesses, restarting S_s would require resending of its operand tuples and so on. To prevent this cascading effect to lead to a complete restart of the algebraic subtree to node n_f, each site can use a *tuple cache*. This cache stores tuples produced by the algebra node furthest downstream on each site. In this way tuples can be resent after site replacement without recomputation and without involving sites further upstream. Caching will of course require storage at each site, but depending on operator this cost can be marginal compared to network cost of a complete restart of the algebraic subtree.

Assuming a failed operation was partially completed, restarting it can produce duplicates. Our approach is to prevent these duplicates from being sent rather than later removing them downstream. Details are given in Section 5.2.

In this section we will assume that operations are mostly read-only, with updates only done in batch at particular times (i.e., similar to assumption doen in previous work as e.g. [6]). This assumption fits well with our intended application areas. In Section 6 we will describe how partial restart can be supported also in the context of updates.

In the rest of this section, we present in more detail the approach for partial restarts of queries.

5.1 Determining Replacement Site

After S_t detects a failure in S_f, it must find a replacement site S_r (denoted $findReplacementSite()$ in the following algorithms). How this is done depends on whether one or more of the algebra nodes at S_f were table/fragment access operations or not.

After S_r has been selected, sites downstream must be notified of the selection so that they can update their local copy of the algebra tree. This is necessary to ensure that handling of subsequent failures are done using an algebra tree with correct node assignment.

Site with no table/fragment access: If n_f is an operator, any site can potentially be selected as replacement (we assume equal capabilities of all sites). Because S_t knows its algebraic subtree(s) and assigned sites, it has enough information to select an S_r that limits network traffic (i.e., the same site as one of S_s).

Site with table/fragment access: If n_f is a local table fragment access, the replacement site must be selected among other sites with replicas of the same fragment. If no live replicas exist, the detector site notifies the initiator site. Application dependent, the initiator can chose to abort or continue with incomplete data.

5.2 Algebra Node Restart

This section describes in detail how different algebra nodes can be restarted. We first discuss the simple case of table access operations. We then turn to operators, which can be categorized into two categories: *stateless* and *stateful* operators. In the case of stateless operators, each tuple is processed completely independent of other tuples. Examples of such operators are select and project. In the case of stateful operators, a result tuple is dependent on several operand tuples. Examples of such operators are join and aggregation.

5.2.1 Table/Fragment Access

Restarting local table fragment access can be performed independent of other sites. After S_s detects a failure of S_f, it must locate a suitable S_r using catalog lookup and transmit the table fragment access operation to it so that the operation can be restarted. To avoid sending duplicates, S_s notifies S_r of the number of tuples it received from S_f before the failure. By assuming deterministic order of sent tuples (for example by acessing the table in the "natural order", either based on row identifier or primary key), this number is enough to ensure that S_r can restart exactly where S_f failed.

5.2.2 Stateless Operators

In the case of stateless operators like select and project, each operand tuple can be processed independently, and the tuple is not needed after it has been processed as long as the results are not lost. Therefore, a full resend of tuples from source nodes is not needed when partially restarting such operators.

For example, if t_i was the last operand tuple processed by a project node before it failed, the replacement project node must start processing on operand tuple t_{i+1}. To know how many operand tuples an operation has consumed, each produced tuple packet is tagged with the last operand tuple(s) used to produce the result tuples in the packet. This can be more than one number if the operand was a fragmented table (one operand tuple number per fragment). This algorithm is shown in Algorithm 2 and explained below.

Algorithm 2 Restart of select or project.

At site S_t, after detecting failure of S_f:
- $S_r \leftarrow findReplacementSite()$
- $t_{ns} \leftarrow tupleNumbers(lastPacket)$
- $Send(N_f, S_r)$ {Algebra subtree}
- $Send(t_{ns}, S_r)$

At site S_r, after receiving N_f and $tupleNum$:
- $n_f \leftarrow root(N_f)$
- $n_s \leftarrow children(n_f)$ {May be more than one node}
- $S_s \leftarrow getAssignedSites(n_s)$
- $Send(S_r, S_s)$ {S_s may be more than one site}
- $Send(t_{ns}, S_s)$

After S_f fails, S_t selects S_r and transmits the relevant algebra subtree (with n_f as root). In order to do a partial restart, S_r must resume select/project where S_f failed. To do this, tuple packets produced by n_f are tagged by the last source tuple number(s) tn_s from n_s used to produce the packet. During recovery, tn_s is transmitted from S_t to S_r and then to S_s. This makes it possible for S_s to resend tuples from tn_{s+1} to S_r. In order for this to work, n_f must produce tuples in a deterministic order (cf. Section 5.3 on how this can be achieved).

When n_f is restarted at S_r it must continue the tuple numbering where S_f left off. I.e., the first packet received at S_t from S_r must have a number higher than t_{ns}. This is necessary to ensure that subsequent failures of n_f is restarted from the correct point.

5.2.3 Stateful Operators

For stateful operators like join and aggregation, each result tuple is dependent on more than one operand tuple. For example, when joining two operands A and B, each tuple from A is matched with each tuple from B. This means that as long as not all tuples from A has been received, all tuples from B are needed. Thus when restarting join, all tuples from B must be resent if the join crashed before all tuples from A had been processed (and vice versa).

Aggregation has similar properties: no results can be produced before the operand has been completely received (at least if we assume no grouping and unsorted tuples). Also, all result tuples are dependent on all operand tuples. So regardless of when aggregation fails (before/during transfer of the result), all operands must be resent to the replacement node.

What can be done when restarting operators such as join and aggregation, is to prevent S_r from sending tuples already sent by S_f before it failed. In order to do this, S_t sends S_r the number of tuples it has received, tn_f. During processing at S_r, these tuples are discarded. This is possible if tuples are produced in a deterministic order – so that the tn_f first tuples from S_r are the exact same tn_f tuples S_f produced first. The algorithms for restart of join and aggregation are presented formally in Algorithm 3 and 4.

We assume a pipelined hash-join algorithm in order to have the join node produce results as early as possible. This allows operators further downstream to start processing as soon as possible and therefore lets more operators execute in parallel.

Algorithm 3 Restart of join.

At site S_t, after detecting failure of S_f:
- $S_r \leftarrow findReplacementSite()$
- $tn_f \leftarrow numberOfTuplesReceived(n_f)$
- $Send(N_f, S_r)$ {Algebra subtree}
- $Send(tn_f, S_r)$

At site S_r, after receiving N_f and tn_f:
- $n_f \leftarrow root(N_f)$
- $n_s \leftarrow child(n_f)$
- $S_{s1} \leftarrow getAssignedSite(n_s)$
- $S_{s2} \leftarrow getAssignedSite(n_s)$
- $Send(S_r, S_{s1})$
- $Send(S_r, S_{s2})$

For joins, it is possible to optimize the algorithm if the failed join node had processed all tuples from one of the sources. Then the processed tuples from the other source do not have to be resent. This is because one can be certain that these tuples have already been joined with all tuples from the first source.

Algorithm 4 Restart of aggregation.

At site S_t, after detecting failure of S_f:
- $S_r \leftarrow findReplacementSite()$
- $tn_f \leftarrow numberOfTuplesReceived(n_f)$
- $Send(N_f, S_r)$ {Algebra subtree}
- $Send(tn_f, S_r)$

At site S_r, after receiving N_f and tn_f:
- $n_f \leftarrow root(N_f)$
- $n_s \leftarrow child(n_f)$
- $S_s \leftarrow getAssignedSite(n_s)$
- $Send(S_r, S_s)$

5.3 Achieving Deterministic Delivery of Tuples

For the algorithms described above, deterministic order of tuples produced by operations is vital. It allows us to use a single tuple number to describe the restart point for the transmissions from the replacement site. Note that deterministic order does not imply that the tuples have to be sorted.

For table fragment accesses, deterministic order can be ensured by, for example, sorting tuples on primary key or by having a local DBMS with deterministic table access operations (which is the general case). For operator nodes, we require that all sites use the same algorithms and that these give deterministic results if they process operand tuples in a deterministic order.

Two issues have to be taken care of in order to achieve deterministic order of operand tuples. First, a single operand can have multiple sources (for example due to table fragmentation). Second, an operator node can have two operands (e.g., join). Both these issues are handled by processing tuple packets from the various source sites in an alternating manner — pausing if a packet from the next source is not yet received. The first packet is taken from the source site with the lowest ID. These IDs are assigned during query planning. Assuming sources can supply tuples with equal rates, alternating between sources rather than processing unordered, should only incur a minor overhead.

5.4 Tuple Caching

After a replacement site S_r has been selected and sent the algebra node n_f, it must notify any sites S_s with source nodes about its existence and request sending of operand tuples. As explained above, the extent tuples have to be resent depends on which operation n_f is. Regardless, S_s may be asked to send tuples produced by algebra nodes long since completed. If means are not taken to prevent it, this can cause a cascading restart of the entire subtree as S_s will have to retrieve its source operand tuples in order to produce the tuples that are to be sent to S_r.

In order to prevent this cascading restart, each site can use a tuple cache where produced tuples are stored until the query is completed and there is no risk of restart. This cache is optional as there is always an option to restart the subtree completely, but as will be shown in Section 7, savings can be large — especially for operations that produce far fewer tuples that they consume (aggregation is a good example). It is therefore possible to have tuple caching of results from some operations and not for others — this can for example be decided during query planning.

Only tuples produced by the most downstream algebra node assigned to a specific site, need to be cached. A tuple cache between two nodes assigned to the same site would in any case be lost when the site fails and would thus be of no use during query restart. Further, tuples from table fragment access operations should not be cached as it can be assumed that they can equally well be retrieved

from the local DBMS. Finally, sites directly upstream from the initiator site need not cache tuples. Such cached tuples would only be of use in case of initiator site failure, but then the query is aborted anyway.

After a query completes, the initiator site notifies all involved sites so they can purge their tuple cache of any tuples belonging to the query. Generally, purging tuple caches during query execution is difficult because of stateful operators such as join and aggregation that upon failure generally requires a full resend of source operands to be restarted.

Since the tuple cache is assumed to be main-memory based, its size will be limited. This means that it is possible that during a query it fills up. When this happens, it simple stops caching new tuples. In this case, we have *partial tuple caching*. During a subsequent failure, the contents in the cache will be used, but the tuples that could not fit needs to be re-created by the upstream operators. Thus the efficieny of the tuple cache during restart depends on the amount of tuples that could fit.

5.5 Handling Multiple Failures

So far, we have assumed that each site participating in a query is assigned only one algebra node and that only a single site fails. In reality, a failed site would likely have been assigned more than one algebra node as this tends to reduce network traffic by having operators use local data. It is also possible, though not as likely, that more than one site could fail. How multiple algebra node failures is handled, depends on whether they occur simultaneously or not.

5.5.1 Non-simultaneous Failures

A new failure after a recovery has completed, requires no extra effort. It only requires that information about the replacement node S_r had been sent downstream so sites could update their algebra trees with the revised site assignment. This makes sure that replacement nodes for later failures get sent a correct algebra subtree.

5.5.2 Simultaneous Failures

If several algebra nodes fail simultaneously (or another fails during recovery), the failed nodes' relative positions in the algebra tree have consequences. With the use of a tuple cache, decentralized failure detection and handling, the effect of a failure is limited to the detector site, the replacement site and its source sites (S_t, S_r and S_s respectively). This makes it possible to handle failures further apart in the algebra tree independently.

When a failed site has two or more directly connected algebra nodes, only the furthest downstream node can be partially restarted as it is the only node where information about results generated before failure, is available. The upstream nodes will therefore have to be completely restarted. This is illustrated in Figure 2. Note that S_s can resend from its tuple cache (if available), limiting the consequences of the failure.

If two sites with directly connected algebra nodes fail at the same time, the upstream failure will not be detected until the downstream failure has been recovered from (due to downstream failure detection). The two failures will therefore be handled one after the other. Because of the upstream failure, its tuple cache will be lost so the upstream node will have to do a complete restart. Again, tuple caches on sites upstream from the upstream failure can be used to reduce network traffic during recovery.

5.6 Cost During Normal Operation

An important aspect of PROQID is a very low overhead during normal operation. The extra cost incurred by using PROQID comes from a few different sources.

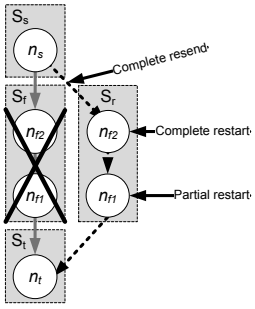

Figure 2: Partial restart with two-node failure.

Operators must produce tuples in a deterministic order. This is achieved by 1) having table scans by rowID (or similar) to retrieve tuples from local storage in a deterministic order (which will in general only incur a minor, if any, overhead), and 2) forcing operators to process input tuples in a deterministic order (whether from the one source or several), and in that way producing tuples in deterministic order. In case of different arrival rates or packets received out of order, processing may be delayed.

There is some network overhead from sending tuple numbers with tuple packets. However, this overhead is minimal if packets contain a few hundred tuples, typically less than 0.1% (depending on tuple size).

Overhead in failure detection is kept low by only detecting failures in active sites and by using regular messages as "I'm alive"-messages as much as possible . It should be noted that any system that supports failure handling has to accept a minor overhead from failure detection.

Tuple caching is an optional part of our approach. Sites can opt to use available memory resources to (partially) cache produced tuples to reduce processing needed in event of partial restart. Since the use of memory for tuple caching can affect other queries, it can incur a significant extra cost. However, use and size of the tuple cache is optional (no cache, partial cache, full cache), and it is trivial to let the user enable the tuple cache for individual queries or query operators when desired.

6. DYNAMIC DATA

The approach presented in the previous section will perform correctly as long as no operations are performed on the databases that can change tuples' ordering in a subsequent restart. Updates are assumed to not change tuples' order, but inserts and deletes of tuples can change the order.

So far, we have assumed a context of read-mostly databases where partial restart can be performed if no inserts or deletes have been performed during the execution of a query. In the case of inserts or deletes between query start and failure, complete restart will have to be performed to get a correct result. Assuming inserts/updates will only be performed occasionally, most failed queries can be handled by partial restart. This can be expected to be the case for typical data warehouse scenarios and also for most of the data to be queried in computational science applications.

In some cases, we also want to be able to support partial restart in application areas where inserts and deletes occur more frequently. Correct results can be achieved in a number of ways:

- Locking: Tables participating in the queries are read-locked so that no updates can be performed. This will delay update transactions performed concurrently with queries, and will

in general only be acceptable with infrequent updates, typically only performed in batch (i.e., no interactive response expected).

- Logging: Inserts/deletes are logged, so that during partial restart the same tuples, and in the same order, as was delivered before restart can be delivered. This solution will essentially provide a snapshot of the database as of the start of the query.
- Tuple cache: As processed tuples can be stored in a site's tuple cache, inserts and deletions in source operands will have no effect unless the contents of the tuple cache are lost. In essence, this works similarly to logging so logging need only be used in sites with no tuple cache (typically local table fragment access).

The methods described above for ensuring correct results have all certain associated costs or they limit concurrency. It is thus important to note that many applications, for example data warehouses, can tolerate a slightly inaccurate results and therefore use partial restart even in case of some inserts and deletions. Inserts can cause duplicate tuples in the result as a tuple numbered $tn-1$ becomes tn after insert and a resend from tn also will include $tn-1$ previously sent. Similarly, deletions can cause missing tuples.

7. EVALUATION

In order to demonstrate the feasibility and query restart cost reduction of using PROQID, we have implemented the approach as described in this paper in the DASCOSA-DB distributed database system. We will in this section describe results from experiments. Each experiment is a distributed query where one of the participating sites fails.

We will first describe the experimental setup, including the implemented prototype and the data set used for evaluation. Two sets of experiments were performed and results are presented in the subsequent sections. The first set contains simple queries that together highlight different aspects of our approach. In order to evaluate PROQID with more realistic queries, the second set contains several complex TPC-H queries. For all queries, we compare the performance of 1) partial restart and 2) complete restart, compared to execution without failures.

Using PROQID instead of complete query restart, we achieve savings in both network communication and query execution time. In order to demonstrate this, we measure savings in terms of *transported tuples* for the simple queries, while we measure time savings for the TPC-H queries.

It should also be noted that using PROQID also gives substantial savings in use of CPU as only affected subqueries have to be restarted. This will result in further query time savings in a multiuser environment.

7.1 Experimental Setup

We have implemented a Java-based prototype, DASCOSA-DB, and extended it with the partial query restart techniques described in this paper. The prototype acts as a middleware on top of Apache Derby running as the local DBMS on each site. DASCOSA-DB accepts queries in standard SQL, transforms them into relational algebra and distributes the algebra nodes to the participating sites after query planning. The catalog service for indexing tables was implemented using the FreePastry DHT.

In our experiments we use data with a schema based on the one used in the TPC-H benchmark (TPC-H). TPC-H is a decision support benchmark reflecting a database scenario similar to our assumed context. The first set of query experiments use the part of the TPC-H schema illustrated in Figure 3, while the second set of queries use the entire TPC-H schema.

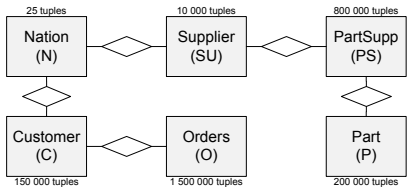

Figure 3: Subset of the TPC-H database.

Site	N	SU	PS	C	O	P
S_0			$PS_{1(1)}$	$C_{1(1)}$	$O_{1(1)}$	$P_{1(1)}$
S_1	$N_{1(1)}$		$PS_{2(1)}$	$C_{2(1)}$	$O_{2(1)}$	$P_{2(1)}$
S_2		$SU_{1(1)}$	$PS_{3(1)}$	$C_{3(1)}$	$O_{3(1)}$	$P_{3(1)}$
S_3		$SU_{2(1)}$	$PS_{4(1)}$	$C_{4(1)}$	$O_{4(1)}$	$P_{4(1)}$
S_4			$PS_{5(1)}$	$C_{5(1)}$	$O_{5(1)}$	$P_{5(1)}$
S_5			$PS_{1(2)}$	$C_{1(2)}$	$O_{1(2)}$	$P_{1(2)}$
S_6	$N_{1(2)}$		$PS_{2(2)}$	$C_{2(2)}$	$O_{2(2)}$	$P_{2(2)}$
S_7		$SU_{1(2)}$	$PS_{3(2)}$	$C_{3(2)}$	$O_{3(2)}$	$P_{3(2)}$
S_8		$SU_{2(2)}$	$PS_{4(2)}$	$C_{4(2)}$	$O_{4(2)}$	$P_{4(2)}$
S_9			$PS_{5(2)}$	$C_{5(2)}$	$O_{5(2)}$	$P_{5(2)}$

Table 3: Table fragments and site distribution.

The queries are performed on a dataset created using the data generator provided at the TPC-H web site. The number of tuples for each table is also shown in Figure 3. The tables used in the queries were fragmented into equally sized parts and distributed to 10 sites as summarized in Table 3. Note that all fragments are replicated on two sites.

Unless otherwise stated, tuple caches were used for all experiments. Caches were made large enough to contain all produced tuples, so cache size was not considered.

In order to crash sites in a deterministic manner, a special crash algebra node was inserted into the generated algebra trees before they were distributed and query execution started. The crash node would let a predetermined number of tuples pass through and then crash the site — stopping the execution of any other algebra nodes assigned to the site.

7.2 Simple Query Results

In this section we present the results from the first part of our experiments. Three different queries were executed; each designed to illustrate different aspects of our approach for partial restart. Each query was executed by DASCOSA-DB and we measured how many transported tuples were needed to complete the query with varying time of failure.

The baseline we compare against is the traditional solution where a query is aborted after failure of one of the sites participating in the query and then completely restarted.

7.2.1 Query 1: Distributed Select

The objective of this test was to investigate the restart performance of stateless operators. We used *select*, but *project* could equally well have been used. The query used was:

```
SELECT * FROM part WHERE p_size < 21
```

This query was transformed to the algebra tree illustrated in Figure 4 (left). Note that we used distributed *select*, i.e., each site with a fragment of **part** performed *select* on its local data. The effects of a centralized *select* are investigated in the next section. The results from the execution of this query are shown in Figure 4 (right).

The x-axis is time of failure computed as TC/T where T is the number of tuples sent from S_2 during execution without failure and TC is the number of tuples sent from S_2 before failure. The y-axis

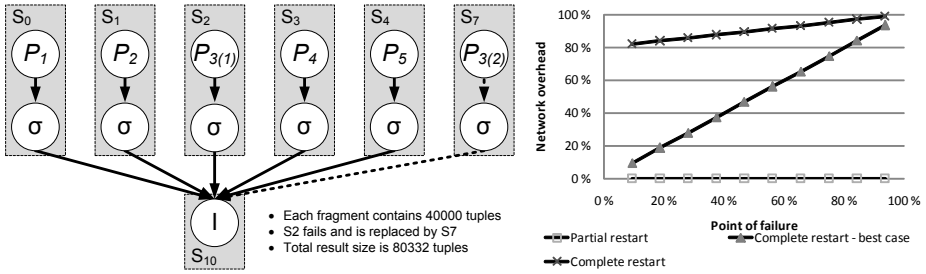

Figure 4: Query 1 and results.

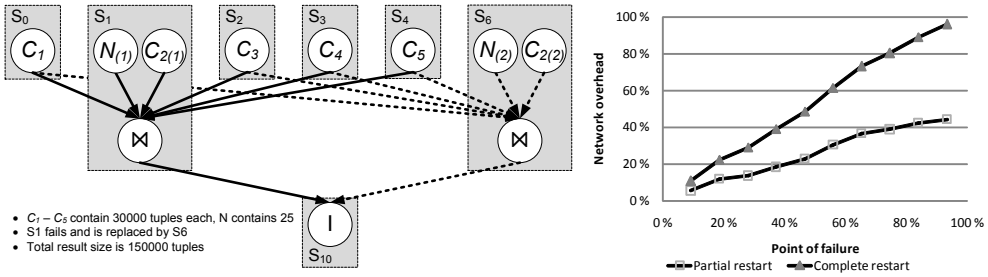

Figure 5: Query 2 and results.

is the extra network traffic needed to handle the failure as NC/N where N is the number of tuples sent by all sites during execution without failure, while NC is the number of tuples sent by all sites during execution with failure.

With complete restart of the query, all data generated before the failure is removed from the participating sites and discarded. Therefore the percentage of extra tuples with complete restart is simply the amount of tuples transmitted before failure. In our example, each *select* produced only 16000 tuples so due to overhead with detecting failure of S_2 (message timeout and alive check) the other sites had sent all their tuples when S_2 was crashed. With larger amounts of data, this overhead would have been reduced and made it possible to stop other sites before they had sent all their tuples. We have therefore plotted two cases of complete restart: The measured values and an estimated best case. The best case assumes that all all sites $S_0 \ldots S_4$ have equal rate of tuple transmission and that all stop as soon as S_2 fail.

With partial restart, S_7 can continue right where S_2 stopped as described in Section 5.2. Nodes S_0, S_1, S_3 and S_4 were unaffected by the failure. Therefore no extra tuples were transmitted for partial restarts compared with no failure.

7.2.2 Query 2: Join

As argued in Section 5.2, stateful operators such as *join* are fundamentally different from *select*. This test case is a simple *join* designed to highlight these differences:

```
SELECT * FROM nation JOIN customer
```

The resulting algebra tree is illustrated in Figure 5 (left). We see that **nation** has two fragments while **customer** has five. Site 1 was selected for the join operator during planning to minimize network traffic as it has a fragment of both involved tables. The results from the execution of this query are shown in Figure 5 (right).

Since *join* is a stateful operator, all source operands must be retransmitted to the replacement site. The only way partial restart is able to improve on complete restart in this case, is to prevent the new join node from transmitting tuples already sent from the failed site. These tuples make up the difference between the two lines in the graph.

In this query, the crashed operator has operands on other sites. Tuples that are received from these sites but not processed before crash, cause the graph lines to deviate slightly from a straight line.

7.2.3 Query 3: Multiple Operators

After investigating the properties of stateless and stateful of operators separately, we made a more complex test case containing a combination of operators:

```
SELECT * FROM supplier JOIN (
  SELECT ps_suppkey, COUNT(*)
  FROM part JOIN partsupp
  WHERE ps_availqty < 1000
  AND p_size < 21
  GROUP BY ps_suppkey).
```

This query was transformed into the algebra tree illustrated in Figure 6. After *select* each fragment of **part** and **partsupp** was reduced to about 16000 tuples. *Aggregation* produced 9624 tuples which was joined with two fragments of **supplier**, each 5000 tuples. The final result was 9624 tuples as well.

With this query it made sense to vary which site we crashed. S_0, S_3 and S_4 all have stateless operators and no external operands and can therefore be treated the same. We first evaluated a crash of S_4. As S_4 has a similar role to S_2 in Query 1, it is not surprising that we got similar results. Partial restart can be done without any extra tuples transferred.

If S_1 is the failing site, all operands to its join must be resent. Unfortunately, as aggregation does not send anything until it is completed, this is an example where the partial restart in effect provides negligible benefits. Beyond avoiding duplicates from the aggregation node, the only saving in our example is that tuples sent from the second **supplier** fragment on S_3 will not have to be resent.

The last site to evaluate failure for is S_2. Execution of algebra nodes on this site has two separate phases. The first phase is local

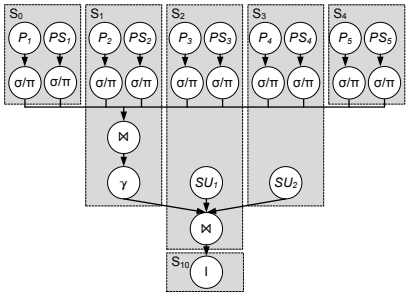

Figure 6: Query 3.

table fragment accesses and *select/project* on the results. Then the site has to wait for the completion of *aggregation* on S_1 before its *join* can start (phase two). Figure 7 (left) shows results where the site fails during phase one, while Figure 7 (right) shows results with failure during phase two.

In phase one S_2 acts as a site without external operands (the *join* has not started). The results therefore match closely, but not perfectly, what we got for Query 1. The reason why partial restart is not optimal is the 5000 tuples transferred from S_3. Depending how early S_2 crash, most/all of these tuples have been received. Since the *join* has not started in phase one, all tuples received from S_3 are unprocessed and must be retransmitted after recovery.

A crash in phase two is a crash during join processing on S_2. As join is a stateful operator, it requires sending of source operands on partial restart. One of the source operands is from the *aggregation* on S_1. Restarting this operator has great consequences as the *join* at S_1 requires the transfer of about 128000 tuples from S_0, S_2, S_3 and S_4. This is a good example of where the tuple cache can be used to great effect. Caching the result of the *aggregation* just requires the storage of 9624 tuples and prevents recomputing *aggregation/join* and resending of their operand tuples. The graph therefore shows partial restart both with and without tuple cache on S_1.

After failure of S_2 during phase two and with the use of the tuple cache, the only tuples that must be resent on partial restart is the contents of the cache and the 5000 **supplier** tuples from S_3. Without the cache, everything must be resent with the exception of tuples already received at the initiator site (S_{10}).

7.3 TPC-H Query Results

In order to validdate our approach in a more realistic setting, we also executed the test queries defined in the TPC-H benchmark. For each query, we measured three execution times:

- T1: Query without failures.
- T2: Query with failure and partial restart.
- T3: Query with failure and complete restart.

Using on these executions times, we calculated *restart cost* as (T2-T1)/(T3-T1) to represent the improvement with partial compared to complete restart. The results from a representative selection of ten TPC-H queries are shown in Figure 8. For six of these queries, the site crashing was executing scan, select and project operators. For the remaing four, the crashing site were also executing more complex operators such as join and aggregation. In two of the queries, the failing site had multiple unconnected algebra nodes.

On average, our approach for partial restart reduces the restart cost of these queries by 50 %. The two queries with least gain (Q13 and Q2) were also the two shortest queries. This is explained by the constant overhead in detecting site failure — which of course plays

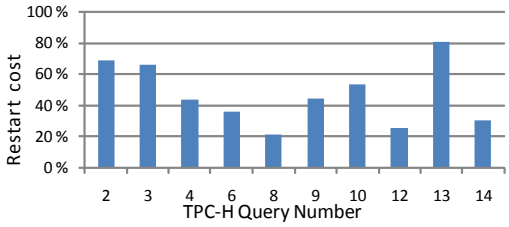

Figure 8: TPC-H query test results.

a larger role for short queries than for long queries. The opposite was also true: the most gain was generally from the longest queries. Finally, restart cost is higher if the site that crashes has blocking operators (aggregation, sorting) rather than non-blocking operators (select, project, pipelined hash-join).

7.4 Summary

The results from the evaluation can be summarized as follows:

- If the crashed site had algebra nodes with no external operands, partial restart can be done without any extra transported tuples compared to query execution without failures.
- For crashed sites with external operands, very few extra tuples must be transported if the failed algebra nodes are stateless operators.
- If failed algebra nodes are stateful operators, partial restart is still better than complete restart.
- For complex queries tuple caching can greatly reduce network traffic during partial restart.
- Partial restart reduces both query execution time and network traffic compared to complete restart.

8. CONCLUSIONS

Distributed database systems are of interest in many contexts where large and time-consuming queries (e.g., Grid databases) or high churn rate (e.g., peer-to-peer databases) are common. Together with a large number of participating sites, this results in an increased probability of failure during query execution. In this paper, we have presented the PROQID approach for partial restart of queries which reduce the overall cost of queries in the presence of failures. The reduction in query cost has been confirmed by experiments using PROQID implemented in a distributed database system.

Future work include the development of cost functions describing the fault-tolerant operators, making it possible for database systems to automatically choose fault-tolerant operators only when conditions in the system (for example network problems) or large queries make it beneficial.

9. REFERENCES

[1] M. N. Alpdemir et al. OGSA-DQP: a service for distributed querying on the Grid. In *Proceedings of EDBT'2004*, 2004.

[2] R. S. Barga et al. Recovery guarantees for internet applications. *ACM Trans. Internet Techn.*, 4(3):289–328, 2004.

[3] P. Bonnet and A. Tomasic. Partial answers for unavailable data sources. In *Proceedings of FQAS'98*, 1998.

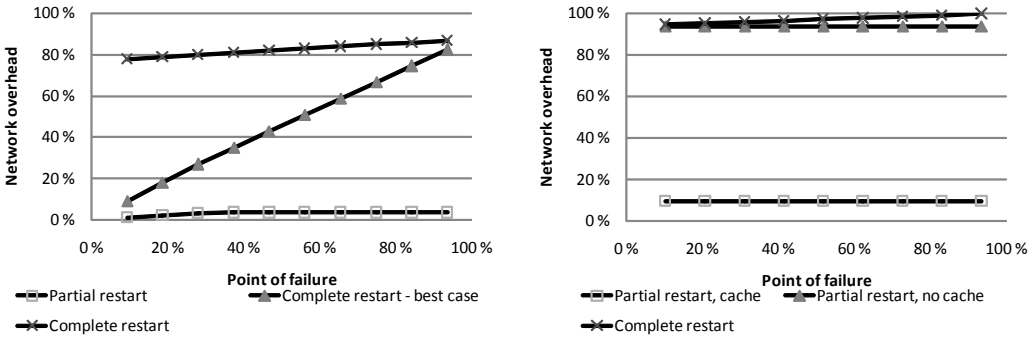

Figure 7: Query 3 results.

[4] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[5] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the SIGMOD'2007*, 2007.

[6] S. Chaudhuri, R. Kaushik, R. Ramamurthy, and A. Pol. Stop-and-restart style execution for long running decision support queries. In *Proceedings of VLDB'2007*, 2007.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE'1995*, 1995.

[8] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB'1996*, 1996.

[9] A. Gounaris et al. Adapting to changing resource performance in Grid query processing. In *Proceedings of DMG'05*, 2005.

[10] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. In *Proceedings of ICDE'2005*, 2005.

[11] J.-H. Hwang et al. A cooperative, self-configuring high-availability solution for stream processing. In *Proceedings of ICDE'2007*, 2007.

[12] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of SIGMOD'1998*, 1998.

[13] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[14] W. Labio et al. Efficient resumption of interrupted warehouse loads. In *Proceedings of SIGMOD'2000*, 2000.

[15] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):192–210, 2003.

[16] A. N. Saharia and Y. M. Babad. Enhancing data warehouse performance through query caching. *SIGMIS Database*, 31(3):43–63, 2000.

[17] J. Smith and P. Watson. Fault-tolerance in distributed query processing. In *Proceedings of IDEAS'2005*, 2005.

[18] R. Wang, B. Salzberg, and D. B. Lomet. Log-based recovery for middleware servers. In *Proceedings of SIGMOD'2007*, 2007.

[19] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.