November 2008

Recovery From Node Failure in Distributed Query Processing

Nicholas E. Taylor
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-38

Recovery From Node Failure in Distributed Query Processing

Abstract

While distributed query processing has many advantages, the use of many independent, physically widespread computers almost universally leads to reliability issues. Several techniques have been developed to provide redundancy and the ability to recover from node failure during query processing. In this survey, we examine three techniques--*upstream backup, active standby,* and *passive standby*--that have been used in both distributed stream data processing and the distributed processing of static data. We also compare several recent systems that use these techniques, and explore which recovery techniques work well under various conditions.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-38

Recovery from Node Failure in Distributed Query Processing

Nicholas E. Taylor

November 12, 2008

Abstract

While distributed query processing has many advantages, the use of many independent, physically widespread computers almost universally leads to reliability issues. Several techniques have been developed to provide redundancy and the ability to recover from node failure during query processing. In this survey, we examine three techniques—*upstream backup*, *active standby*, and *passive standby*—that have been used in both distributed stream data processing and the distributed processing of static data. We also compare several recent systems that use these techniques, and explore which recovery techniques work well under various conditions.

Contents

List of Figures

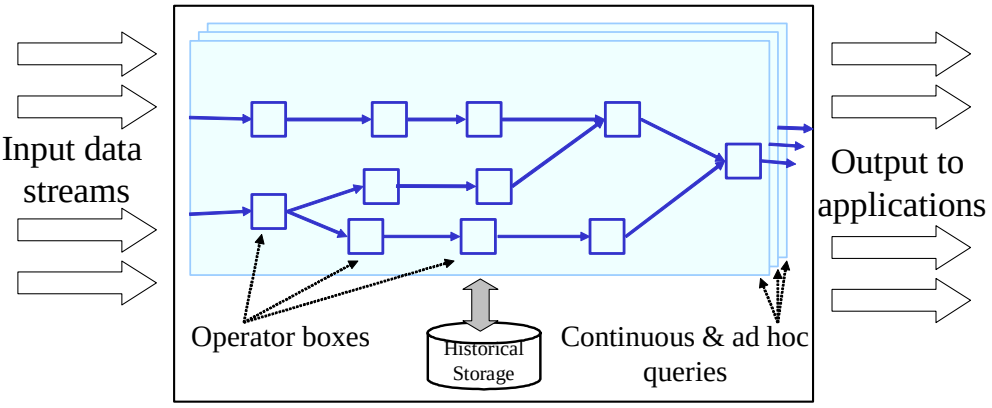


Figure 1: Aurora stream processor overview from Carney et al. (2002).

1 Introduction

The field of distributed databases dates back almost as far as the entire database field. As described in Lindsay (1987), work on R* dates back to 1979, and Rothnie et al. (1980) marks the first published results on SDD-1. Reliability as it relates to distributed transaction processing is well-studied, and is described in such textbooks as Özsu and Valduriez (1999, chap. 12) and Bobak (1996, chap. 11). However, such techniques do not address what happens when a failure occurs during a query instead of an update. If the distributed data is replicated, the system can simply restart the query using a new set of nodes; however, this may be inefficient if the query is nearly complete as all incremental results are lost. Furthermore, if the system is executing continuous queries over streaming data (Balakrishnan et al., 2004; Abadi et al., 2005; Chandrasekaran et al., 2003), it may not be possible to restart the query and restore the operator state that was on the failed node. Both of these problems have motivated recent work on failure recovery. These approaches build on existing techniques based on replication, as described in, for example, Helal et al. (1996, chap. 3), but with database-specific optimizations to improve performance.

In this survey, we compare approaches used in a number of distributed database systems. The systems we examine are quite diverse. Some support streaming data, some support static data. Some support *partitioned parallelism* through some implementation of the Exchange operator (Graefe, 1990), and others do not. This spreads data for a single logical operator across many physical nodes based through some deterministic splitting algorithm, typically using range- or hash-based partitioning. All support pipelined parallelism, where different operators are located on different nodes.

In the database community, *stream processing* generally refers to query processing over unbounded streams of input tuples. These tuples pass through operators similar to those in a traditional database system. Some operators are identical, while others are modified to better suit the streaming data paradigm. Joins, for example, are typically conducted within a window (specified either in time or number of tuples) to avoid buffering an entire data stream. Aggregation operators periodically emit results or return incremental results, and also typically have windowed semantics to avoid large amounts of buffering. Figure 1 shows a diagram of Aurora (Carney et al., 2002; Balakrishnan et al., 2004), an early stream processing system. More recent systems generally have a similar design, but

allow the processing to take place in a distributed fashion. Stream processing has a variety of uses; examples from the literature (Balazinska et al., 2008) include sensor monitoring, network intrusion detection, stock ticker processing, and many others.

Hwang et al. (2005) defines three kinds of recovery modes for stream processing. *Precise recovery* produces the exact same output after a failure as it would have had there been no failure, though clearly the results may necessarily be delayed (though not reordered) due to the extra processing needed to recover from the failure. *Gap recovery* may skip some results before producing the remainder of the results that would have been produced without the failure. *Rollback recovery* is somewhat more complex, and its precise effects depend on the query plan chosen; in contrast, gap and precise recovery are only defined in terms of the logical definition of the query. Rollback recovery returns the system to a state before the failure and then resumes execution. While Hwang et al. (2005) examines recovery techniques in the context of rollback recovery, all of the other systems we study enforce additional guarantees to ensure that these same techniques produce precise recovery. It might also seem logical to define a recovery mode that allows tuple reordering. However, given that some stream operators (the windowed join and the windowed aggregate) are defined in terms of tuple windows, such a relaxation at an intermediate operator could change the actual results at a later operator, and not merely their order.

Furthermore, Hwang et al. (2005) introduces three general approaches to recovery in stream processing. The techniques used in the systems we examine all fall into one of these three categories. Perhaps the simplest approach is *upstream backup*. Upstream nodes act serve as backups for downstream nodes by retaining a copy of their output until it has been processed by their downstream neighbor and sent on to another node. In this way, there are multiple copies of each tuple available, providing redundancy. We examine this technique in a distributed query processing engine in Section 2. The remaining two approaches are closely related to previous work on process pairs (Gray, 1985; Bartlett, 1981). *Active standby* creates two (or more) copies of each operator exists, and they all process the same data in the same order (as in so-called *lockstep* process pairs). Failover from one copy to another is relatively seamless; the system must merely ensure that the operator downstream form the replaced node doesn't process any data more than once and doesn't miss any data. We discuss this in the context of several stream processing engines in Section 3. This final approach, *passive standby*, replicates each operator onto two (or mode) distinct nodes. The primary node actually performs the computation. Periodically, it sends a checkpoint of its operator state to the secondary node or nodes. If the primary node fails, the secondary node can take over using the checkpoint, as in *checkpointing* process pairs. Any tuples processed by the primary node but not yet checkpointed will also need to be reprocessed; this entails a small amount of buffering at upstream operators. We discuss the use of this approach in a stream-processing system in Section 4. Following a discussion of these systems, we will conclude with a comparison of these approaches in Section 5, based on both simulation results from Hwang et al. (2005) and experimental results from Hwang et al. (2007).

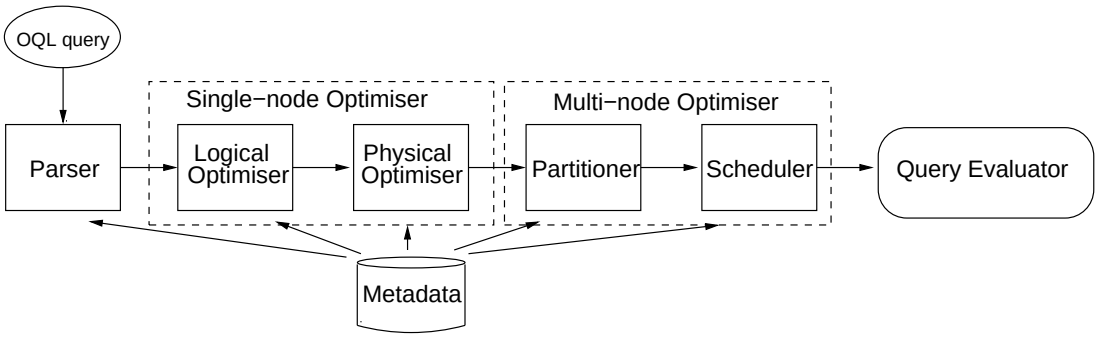


Figure 2: Architecture of the OGSA-DQP system

2 Upstream Backup

As described above, upstream backup stores redundant copies of in-flight data at processing nodes as they pass through the operator tree. In this way, each node acts as the backup for data that is currently being processed at nodes downstream from it. If one of them fails, the state stored at upstream nodes can be used to recreate operator state and to ensure that no tuples are lost. Some sort of coordination is used to ensure that only a few copies of each tuple are stored in the system, and then none are stored after they are no longer needed. Clearly upstream backup leads to increased memory use, as intermediate results are buffered to provide fault tolerance, but in the absence of node failure no computational effort, except for coordination to remove intermediate results that are no longer needed. However, recovery time can be expected to be high, as large amounts of data may need to be sent across the network during the recovery process.

2.1 Use in OGSA-DQP

Smith et al. (2000, 2002) describe a distributed query processing engine (OGSA-DQP) that operates over the Open Grid Services Architecture (OGSA), a distributed computing architecture based on Web services.[1] Specifics of OGSA-DQP are not germane to a discussion of resilience to node failure, except to say that it supports both partitioned and pipelined parallelism, and has a standard two-phase query optimizer that transforms OQL queries into parallelized execution plans. Figure 2 gives a brief overview of the system architecture.

An extension to OGSA-DQP, described in Smith and Watson (2005), adds a fault-tolerance mechanism based on upstream backup. Tuples flow through the system in an ordered stream, and interspersed with them are checkpoint markers, as shown in Figure 3. Each exchange operator (shown in Figure 4) introduces checkpoints into the tuple stream, and forwards checkpoint operators from other nodes along. It buffers the data it sends into its recovery log, grouped by sequence numbers from the checkpoint markers it adds. When a node receives a checkpoint marker from another node, it decrements the hop count h. If h has gone to zero, it then sends an acknowledgment for that

[1] For more information on OGSA, see `http://www.ggf.org/documents/GFD.80.pdf`.

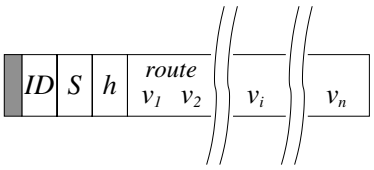


Figure 3: OGSA-DQP Checkpoint Marker. *ID* denotes the node that created the marker, *S* is a node-specific sequence identifier, *h* is the number of hops remaining before the checkpoint is returned, and $v_1 \ldots v_n$ give the route that the checkpoint marker has taken through the query graph.

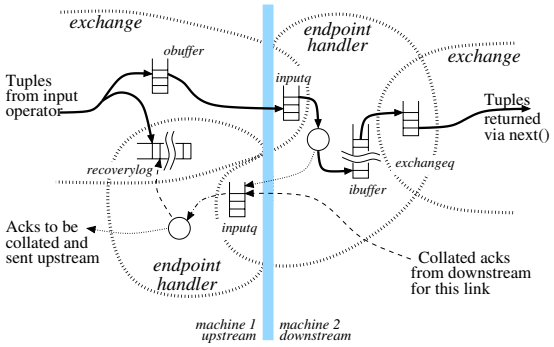
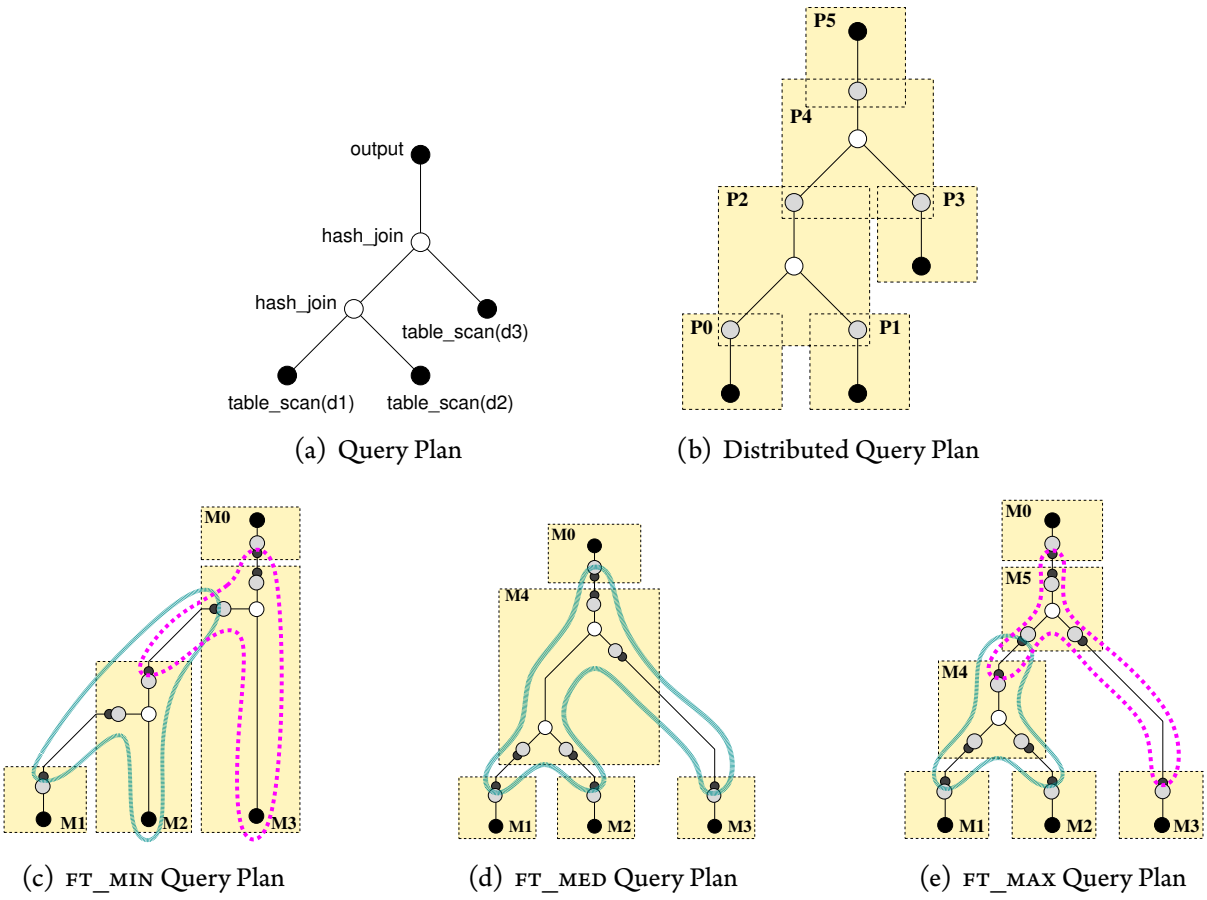


Figure 4: Exchange operator in OGSA-DQP system. Dashed lines indicate thread boundaries.

checkpoint upstream (using the vector v from the checkpoint marker); otherwise it puts the modified checkpoint marker back into the tuple stream. The acknowledgment is forwarded upstream until it reaches the node *ID* that produced it; *ID* can then clear the entry for *S* in its recovery log. In this way, the initial value of *h* determines how many replicas of each data point are retained in the system.

As described, this techniques works well for one-input operators. However, two-input operators (e.g. join, union) are common in real query processing systems. For join, the authors propose to ensure that the smaller input is the left input to the join. That input is fully processed into a hash table before any input from the right side is processed. Then, right checkpoints are processed as they come in, and left checkpoints are not returned until the right input has finished. In this way, the recent part of the right input and all of the left input remains buffered upstream. Clearly this does not work well if both inputs are large, or if it is not known *a priori* which one will be larger. Similarly, aggregate operators (not discussed at all) will need to buffer all of their input.

Recovery in this system is relatively straightforward. A central coordinator determines which nodes have failed, based on data from all participating nodes. It assigns new nodes to take over for failed ones, and installs the query plan on them. When a new node joins the system, it receives the contents of its downstream nodes recovery log (i.e. the data that had been sent to the new node but not yet processed by the rest of the system). This brings the replacement node back the the state that the failed node had shortly before it failed. Since operators and sequence numbers are deterministic, duplicate elimination simply consists of discarding partial blocks (i.e. the tuples that came after the last checkpoint marker) and skipping blocks that have already been processed.



(a) Query Plan (b) Distributed Query Plan

(c) FT_MIN Query Plan (d) FT_MED Query Plan (e) FT_MAX Query Plan

Figure 5: Plans for a simple 3-way join query in OGSA-DQP

An interesting aspect of this work is the discussion of varying the query plan to improve resilience to failure. Figure 5 shows an example query plan and various ways it can be divvied up among different nodes. Resilient plans FT_MIN, FT_MED and FT_MAX are shown in Figures 5(c), 5(d), and 5(e) respectively. In general, the authors argue that increasing the number of nodes improves resiliency by increasing the amount of buffered state available for recovery; for example, if the top join fails, in FT_MAX it can recover its left input from the state buffered at other nodes, eliminating the need to recompute the lower join. It should be noted, however, that such an argument depends on the relative costs of data transmission over the network, query execution, and recovery time. While the authors acknowledge that query optimization for reliability needs to be done in a cost-based way, that is deferred to future work and is not studied.

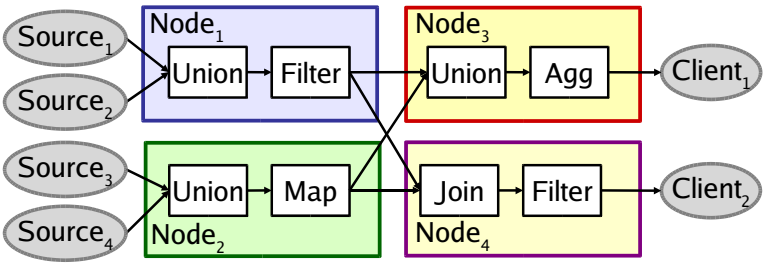
The experimental results show, not surprisingly, that the presented recovery technique is faster that simply restarting the query over a new set of nodes; for replicated, static data this is always an alternative recovery technique. However, the scenario for the main experiment also demonstrates one major shortcoming of this work. The experiment assumes a network where one node experiences a hardware failure, is rebooted, and then can rejoin the network to complete processing. However, this entails a long delay while waiting for the failed node to return. Obviously this could be prevented by the use of a *hot spare* but they do not consider this scenario. Another node could take over for the failed node while maintaining its current tasks as well, but this node would likely become overloaded and be a performance bottleneck. Splitting the work of the failed node over the remaining nodes is not possible due to system design (the correspondence between checkpoint numbers and tuples would change, for example), which is a major limitation. Other systems, such as Flux (discussed in Section 3.2) address this through virtualization, where each physical node executes a number of virtual nodes independently; such an approach would work here as well. The virtual nodes from a failed physical node can be redistributed evenly over all of the remaining nodes while maintaining the identical execution properties needed for failure recovery. The authors do not address such a technique, though if results from Flux are generalizable it would work quite well.

3 Active Standby

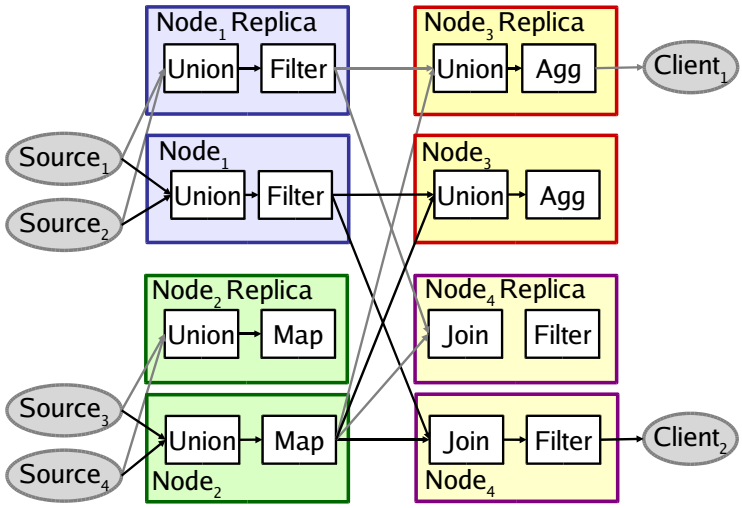
Active standby, as mentioned in Section 1, maintains multiple copies of each logical processing node. The primary copy performs the computation that creates the system output in the absence of failure. The secondary copy (or copies, though typically only one is used) shadows the primary copy by performing exactly the same computation on exactly the same inputs. It is therefore ready to take over almost immediately to replace a failed primary node. However, this approach reduces the amount of processing power available to the system by performing significant redundant computation, even in the absence of any node failure.

3.1 Use in Borealis

The Borealis system, introduced in Abadi et al. (2005), is a "second-generation" stream processing engine that adds a number of features to the earlier Aurora system (Carney et al., 2002; Balakrishnan et al., 2004), including revisions to previous results, and query modification at run-time. For the



(a) Distributed



(b) Distributed and Replicated

Figure 6: Diagrams for a sample query in Borealis

purposes of this discussion, however, the salient feature is the addition of distributed execution. Streams in one query can originate at different nodes, pass through many other sites for processing, and produce results which are sent to yet other sites. Borealis supports pipelined but not partitioned parallelism, meaning that, in the absence of redundancy to provide fault-tolerance, exactly one node will execute each physical operator

Balazinska et al. (2008) details the *Delay, Process, and Correct* (DPC) technique used to provide fault-tolerance in Borealis. The approach replicates each operator so that there are at least two (and possibly more) copies of each operator performing the same computation. Figure 6(a) shows a sample query in the Borealis system, and Figure 6(b) shows a replicated instantiation of the same query. A novel aspect of this approach is that each query must specify the *maximum incremental processing latency* it can endure; that is to say, the maximum amount of time that the system can try to recover from a failure before some attempt must be made to produce results, even if they may later prove to be incorrect. To this end, Borealis distinguished between normal tuples, deemed to be *stable*, and *tentative* tuples. Such tuples are created when an operator is receiving tentative tuples from another operator, or when one of its inputs has failed but it is still receiving input from another

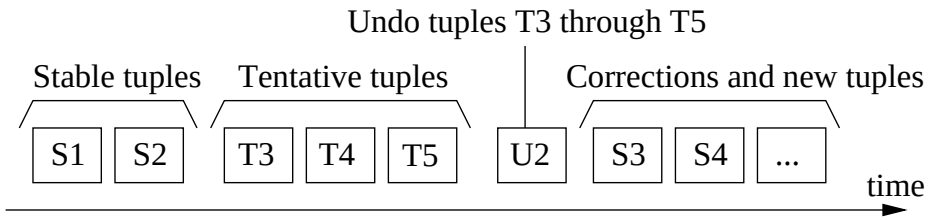


Figure 7: Borealis' use of tentative tuples

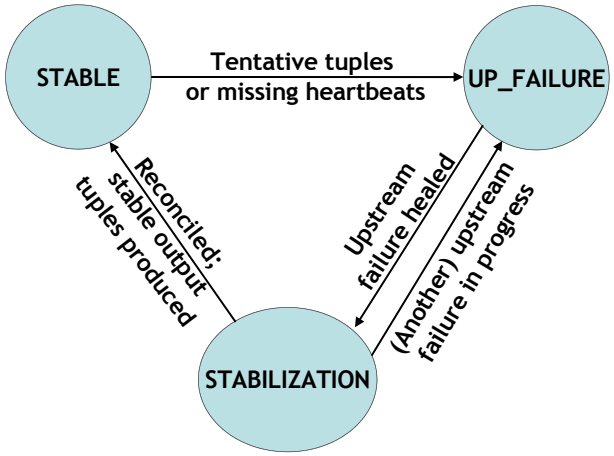


Figure 8: Borealis' DPC State Machine

operator. When an operator recovers from failure, it undoes the tentative tuples before it sends along corrected versions and begins to process new tuples. This process is shown in Figure 7. The rationale is that, in many applications, seeing incomplete data is better than seeing no data at all, as would happen if the system simply blocked until recovery was complete.

The DPC technique depends on replicas of an operator producing the same output based on the same input. Since in-order communication is provided by the underlying network layer, this is straightforwards for single-input operators. To accommodate multiple-input operators like join and union, where differing processing speeds or network latencies between nodes may cause inputs to arrive with different interleavings at different replicas, all tuples are tagged with a timestamp when they are produced. Each stream is periodically punctuated by a *boundary tuple* which indicates that all subsequent tuples will have timestamps strictly greater than that of the boundary tuple. A *SUnion* operator is responsible for serializing the input to each multiple-input operator in a deterministic way based the timestamps and releasing a batch of tuples when all input streams have sent a boundary tuple for a particular timeframe.

Figure 8 shows the states that operators transition between during and while recovering from a failure. All operators start out in the STABLE state. If they encounter an upstream failure, they transition into the UP_FAILURE state. In this state, as soon a STABLE replica of the upstream neighbor is located, the node switches to the STABILIZATION state, which we discuss momentarily. Until then, if the upstream node has failed, it searches for a replica of it in UP_FAILURE state, and switches to

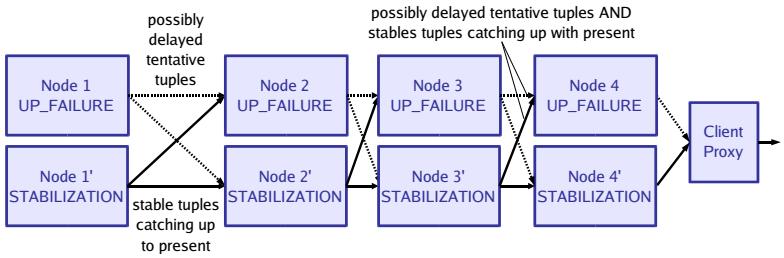


Figure 9: DPC reconciliation in a chain of nodes

that, if one is found. While waiting for a stable replica to appear, it continues to produce tentative tuples based on the data it is receiving. If its upstream neighbor starts to produce corrections (i.e. an undo followed by stable tuples), it also enters STABILIZATION to begin propagating the effects of those corrections downstream.

When an operator enters STABILIZATION, it needs to purge the effects of all tentative tuples from its state and process all of the stable tuples that it has missed before it can begin to process new incoming data. This can be accomplished in a number of ways, but the authors have found that checkpointing, wherein the operator takes a checkpoint of its state upon receiving the first tentative tuple, performs best in most cases. The checkpoint is then restored before processing of stable data resumes. Before processing the new data, the operator also sends an undo tuple (if needed) to cancel out the downstream effects of its tentative output. The node then sends the last stable tuple it received to the new upstream replica, which begins replaying its output at that point; this entails unbounded logging at all nodes, which is a limitation that we will see addressed in other systems. Processing can then resume, and the node will reenter the STABLE state when it has caught up with the upstream node; alternatively, an additional failure will send it back to UP_FAILURE.

There is a complication to this, related to the maximum latency requirement. As outlined above, the recovery technique will suspend output while restoring the operator state. If the checkpoint restoration and replayed data processing is going to take a long time, it may cause an unacceptable delay before the downstream node receives current data. The authors solve this by having a replica of the operator in the UP_FAILURE state send data on to the node downstream from the node that is recovering (if there is another node already in a STABLE state, the stabilizing node should have no downstream node at all, since all operators will attempt to use a stable replica if one is available); this allows the stabilization to take place without causing the downstream node to stop sending data. The downstream node processes the corrections from the stabilizing node and the tentative tuples from the UP_FAILURE node in parallel, producing new tentative tuples while correcting its input stream from the stabilizing replica.

The authors also present an optimization to this technique. The intuition is the following: if a stable replica will be available soon, it is more efficient to delay for a short time computation and then resume using the stable replica, rather than produce tentative tuples which will be immediately undone. To this end, the system can choose to delay tuples for up to the maximum incremental latency in attempt to reduce the number of tentative tuples.

The authors also conduct a series of experiments in which they take the union of multiple data sources and pass the results through varying numbers of nodes while controlling the length of a

temporary failure at one of the sources. The results of these experiments show that, for a single processing node, delaying works well; however, for more complex networks of nodes delaying is only helpful for short failures. This is because the way reconciliation works in a chain of nodes, as shown in Figure 9. Because a node enters the STABILIZATION state as soon as a small timeframe becomes stable (i.e. has been corrected and boundary tuple has been received), all of the nodes that are chosen to stabilize (instead of continuing to produce tentative tuples) do so almost simultaneously. This allows reconciliation to occur in a pipelined fashion, meaning that as the number of nodes in the system increases, stabilization time only increases slightly. Since the number of tentative tuples is therefore proportional to the sum of the failure time and the stabilization time less the delay at the final node, as the failure time increases, the relative benefit of delaying decreases. As delaying increases latency regardless, it therefore become a poorer chose for large delays. A more thorough discussion of this can be found in the paper. The authors conclude that a brief initial delay, followed by immediate processing afterward, is the best strategy to balance the number of tentative tuples and overall system latency.

3.2 Use in Flux and TelegraphCQ

TelegraphCQ (Chandrasekaran et al., 2003) is another recent distributed stream processing engine. The implementation details of TelegraphCQ are not relevant to this survey, except to note that, unlike Borealis, TelegraphCQ supports partitioned parallelism. Flux, as described in Shah (2004), is the Exchange operator for the TelegraphCQ system. As it implements all inter-node communication in the system, it is able to provide fault-tolerance and load-balancing capabilities for the system. Shah (2004) describes three variants of the Flux operator. Flux-HA provides fault tolerance, Flux-LB provides load balancing, and the regular Flux operator provides both. While Flux-HA is simpler than the full Flux operator, the latter offers better flexibility and performance. In this survey, we first present Flux-HA, and then describe full Flux.

All of the replication is Flux assumes one primary and one secondary copy of each operator (or each partition of each operator); this is unlike Borealis, which supported an arbitrary number of replicas. In Flux, as in Borealis, each copy of an operator operates in lock-step with with its counterpart, producing exactly the same output based on exactly the same input. Each tuple is assigned a unique *sequence number*, as it enters the system. As tuples pass through the system, one-to-one transformations leave the sequence number unchanged. One-to-many and many-to-one operators must ensure that they produce tuples with sequence numbers in ascending order. When multiple tuple streams are combined (i.e. by a union or join), the system always processes them in ascending order (by blocking until the next tuple can be read from each input). This ensures deterministic evaluation even when streams are interleaved differently, as in Borealis.

The core of Flux is its *Abstract Kernel Protocol*, shown in Figure 10. The primary and secondary copies operate on the same data but out of sync with each other. As tuples are produced by the primary copy of an operator, they are sent to their consumer. The consumer then sends an acknowledgment (the tuples' sequence numbers) to the copy. The copy, meanwhile, stores produced tuples in its buffer. As it receives acknowledgments from the consumer, it discards the corresponding tuples from its buffer (or, if the tuple has not yet been produced by the copy, records that it doesn't need to save that tuple). If the consumer remembers the sequence numbers of the tuples it has received, it

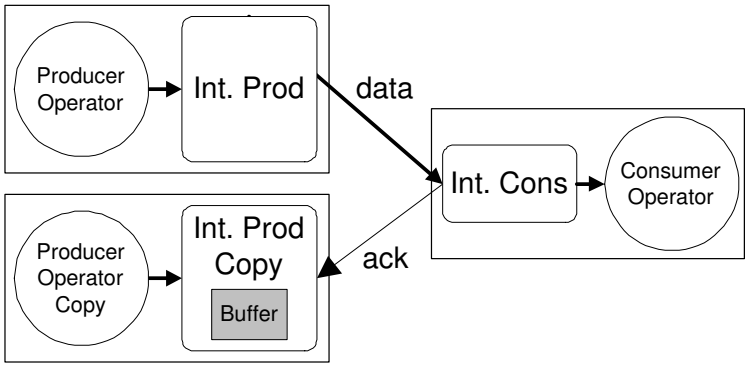


Figure 10: The Flux Abstract Kernel Protocol

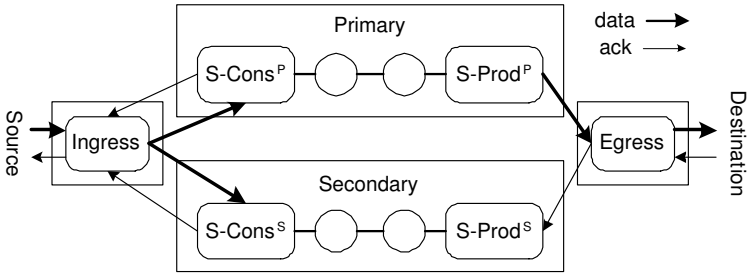


Figure 11: Flux-style redundancy for an entire dataflow

can then construct a loss-free and duplicate-free version of the tuple stream if either the primary or the secondary input fails. Figure 11 shows how this would work for a simple streaming query where the data is received from a source at one node and processed by two other nodes, and the result is then sent to a third now.

Figure 12 shows how Flux Abstract Kernel Protocol works in a partitioned-parallel dataflow. In such a dataflow, the FH-Prod and FH-Cons operators function as the Exchange operator to move data between partitions. To provide redundancy, there are two copies of each partition; in fact, there are two copies of the entire dataflow, connected only at the data source and at the destination of the query results. Each node belongs to either the primary dataflow or the secondary dataflow. Unlike in the simple streaming example above, the data may be sent over the network multiple times. Therefore both the primary and the secondary copies of each partition send data to other partitions, and receive acknowledgments from the counterparts of the partitions they send data to. As long as at most one replica of each partition fails, the system can recover the missing state and produce correct results.

The exact details of the recovery process are not interesting for the purposes of a general discussion of recovery techniques. They are similar in spirit to Borealis, though simpler since there is no desire to produce tentative tuples while recovery is going on; only stable tuples are ever produced. The author also describes how to create a new primary or secondary partition of an operator to replace a failed one. At a high level, this is done by taking a snapshot of the surviving copy and installing it on a new node; the copy can then continue in place of the failed node, though care must be taken

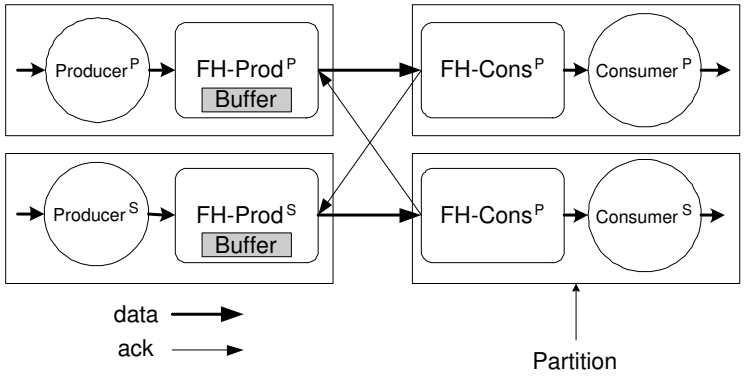


Figure 12: Redundant partitioned parallelism in Flux. The arrows show the paths a particular tuple would take through the primary and secondary dataflows. In general, tuples from one partition in the produced could end up at *any* partition of the consumer.

to avoid duplicate or dropped tuples, perhaps using data from the output buffers in surviving nodes.

As described, this approach (termed Flux-HA in the paper) only works at the granularity of entire nodes. This is not ideal, since a spare node may not be available to replace a failed node, and "doubling up" logical nodes on a physical node may well create a performance bottleneck. In the actual implementation, Flux combines the fault-tolerance techniques described above with load-balancing techniques to make the system more flexible. Each physical node hosts a number of virtual nodes, over which the parallel operators are partitioned. In the load-balancing work, the virtual nodes are moved between physical nodes to improve performance. In the combined Flux, the failed virtual nodes can be distributed over the remaining physical nodes, and then load-balanced to give better performance.

3.3 Comparison of Borealis and Flux

There are several salient differences between the fault-tolerance mechanisms of Borealis and Flux. The largest difference, and the source of much of the added complexity of the Borealis technique, is that Borealis attempts to produce tentative results in the event of a network partition or other temporary failure; Flux will switch to the backup, if available, but otherwise will simply block. The simplified redundancy system does, however, allow Flux to perform automatic buffer management to clear logs when the information in them is no longer needed; Borealis, at least as described in the paper, holds all logged information indefinitely. Additionally, Borealis does not support partitioned-parallel query plans; this limits the ability of the system both to perform load balancing as part of recovery (since an operator may be placed only at one node, instead of split between many).

4 Passive Standby

Like active standby, passive standby has a primary node and one or more secondary nodes for each operator. Unlike active standby, only the primary node performs the main computation. Periodi-

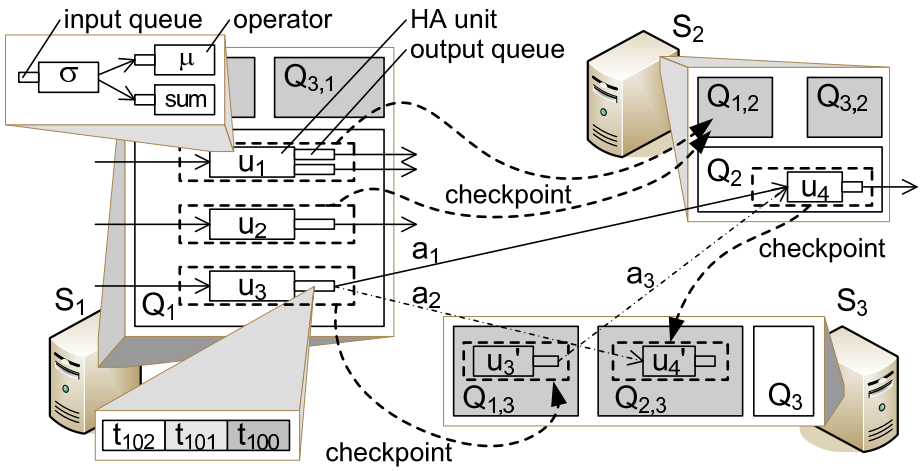


Figure 13: The High Availability state checkpointing model added to Borealis by Hwang et al. (2007)

cally, during processing, the primary node takes a checkpoint of its state, which it disseminates to the secondary nodes. With this state, and the buffering of any uncheckpointed tuples (similar to upstream backup) at upstream nodes, a secondary node can take over for a failed primary node and produce identical results. The simulation results in Hwang et al. (2005) for checkpointing, which we will discuss in more detail in Section 5, were not favorable. However, more recent work has developed sophisticated techniques to improve passive standby performance and make it a reasonable recovery option, one that combines the low recovery time of active standby with the low overhead of upstream backup. In this section, we detail two complimentary approaches to improving performance in the Borealis distributed stream processing engine using passive standby and checkpoints.

4.1 Backup Management

Hwang et al. (2007) discusses techniques to improve performance in Borealis. The operators at each node are grouped into HA *(High Availability)* units, which are the level at which replication is done. Each HA unit consists of the operators, the operators' input queue, and the operators' output queues. This is shown in Figure 13, where HA Unit u_1 contains a filter operator σ, a map operator μ, and an aggregator (sum), the input to σ, and the output of μ and sum. Each HA unit is checkpointed onto a backup server when the server that hosts the HA unit has some idle time; in Figure 13, u_1 is checkpointed from S_1 to S_2. The checkpoint includes all state inside the HA unit, including input and output queues, expressed as a delta from the last checkpoint. Additionally, tuples stay in the output queue of a HA unit until their effects in the node they have been sent to are checkpointed. This guarantees that a single node failure can never cause data loss.

The work provides adaptive techniques for assigning backups to nodes, and for determining when HA units should be split and merged. It also presents two algorithms, *round-robin* and *min-max*, that determine when to take checkpoints and when a node that hosts a checkpoint should merge a delta checkpoint it has received into its copy of the HA unit's state. It is worth repeating that such

actions only take place when a node is idle; the authors assume that overall system capacity exceed the requirements of the query workload and therefore there checkpoints can happen relatively frequently during processing. A detailed discussion of these techniques is well beyond the purview of this document. Briefly, however, round-robin simply cycles between available tasks. Min-max attempts to minimize the recovery time for the HA unit that currently has the maximum recovery time; this could entail either taking a new checkpoint or processing previous delta checkpoint data into the replicated operator state.

Figure 14 summarizes the experimental results for HA checkpointing. The *whole checkpointing* referenced by the graphs simply copies the entire state of a node onto a new server. Figure 14(a) shows that their min-max scheduling algorithm gives good performance over the duration of the experiment, unlike the other approaches, and Figure 14(b) shows that it does not increase processing latency much beyond the baseline of no redundancy.

4.2 SGuard: Improving Checkpointing Performance

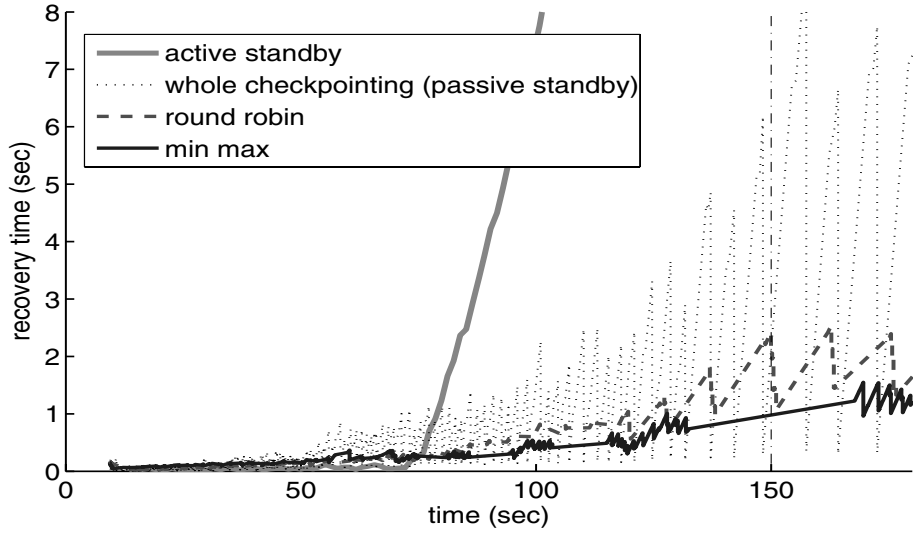
Kwan et al. (2008) presents several complimentary techniques to improve checkpointing performance in Borealis, which the authors collectively term *SGuard*. The model they assume is similar to that used in by Hwang et al. (2005) and described in Section 4.1. An overview of a query in their system in shown in Figure 15; it shows HA units, as before, this time each bracketed by a pair of *HAInput* and *HAOutput* operators.

While the overall technique used, passive standby, is similar to the approach taken in previous work, the way the checkpoints are created and stored is vary different. SGuard stores the checkpoint in a distributed, replicated file system (DFS), such as the Google File System (Ghemawat et al., 2003) or the Hadoop Distributed File System.[2] Such systems are designed to store and retrieve large volumes of data quickly, and are well suited to append-heavy workloads, like storing checkpoints. The authors develop a new scheduler for a DFS to support their system, but the topic is not relevant to the discussion at hand. The use of a DFS does, however, allow reuse of well-designed and optimized external components in place of custom code, and increases the flexibility of the system by allowing arbitrary numbers of replicas of HA unit checkpoints simply by changing a DFS parameter.

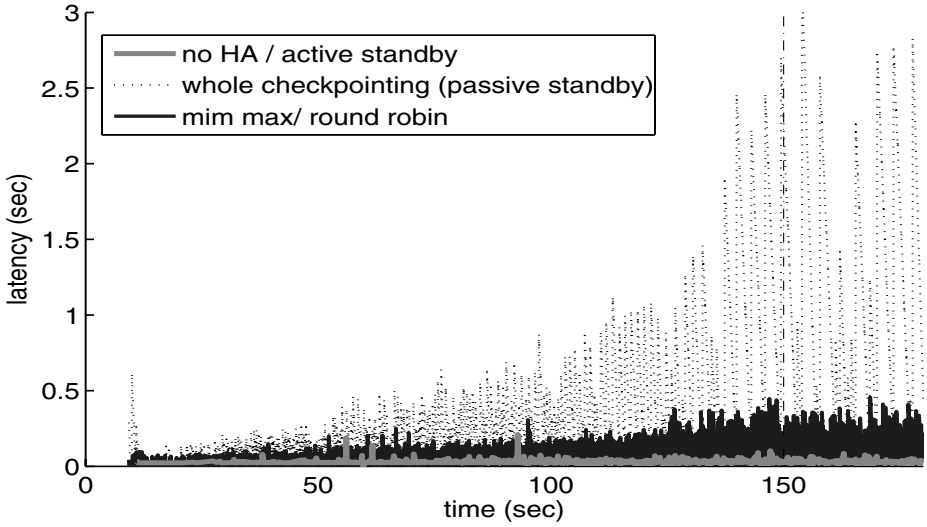
A second optimization is the way operator checkpoints are created. The HA work described in Section 4.1 used custom serializers and deserializers to store and restore operator state. SGuard instead uses a *Memory Management Middleware* (MMM) to checkpoint operator state directly. All operators store all of there state in the MMM, which consists of raw blocks (pages) of memory. Utility methods provide structured access to the raw pages, which is what the operators use to access their internal state. MMM provides two key advantages. First, since data is already serialized, creating a checkpoint entails only writing the MMM pages directly to disk. Second, processing can continue during checkpoint creation through the use of copy-on-write pages; these are implemented within the application level, and do not make use of operating system VM features.

The authors do not compare their checkpoint storage techniques with those presented in the HA work, so we cannot verify the performance benefits of using a DFS to store the checkpoints; how-

[2]For more information on HDFS, see `http://hadoop.apache.org/core/docs/current/hdfs_design.pdf`.



(a) Recovery Time



(b) Processing Latency

Figure 14: Experimental results for HA checkpointing in Borealis from Hwang et al. (2007). Input to the aggregate operators increases from zero tuples/second initially to 2000 tuples/second at time 150 seconds. Active standby cannot cope with the increased load beyond approximately 1000 tuples/second.

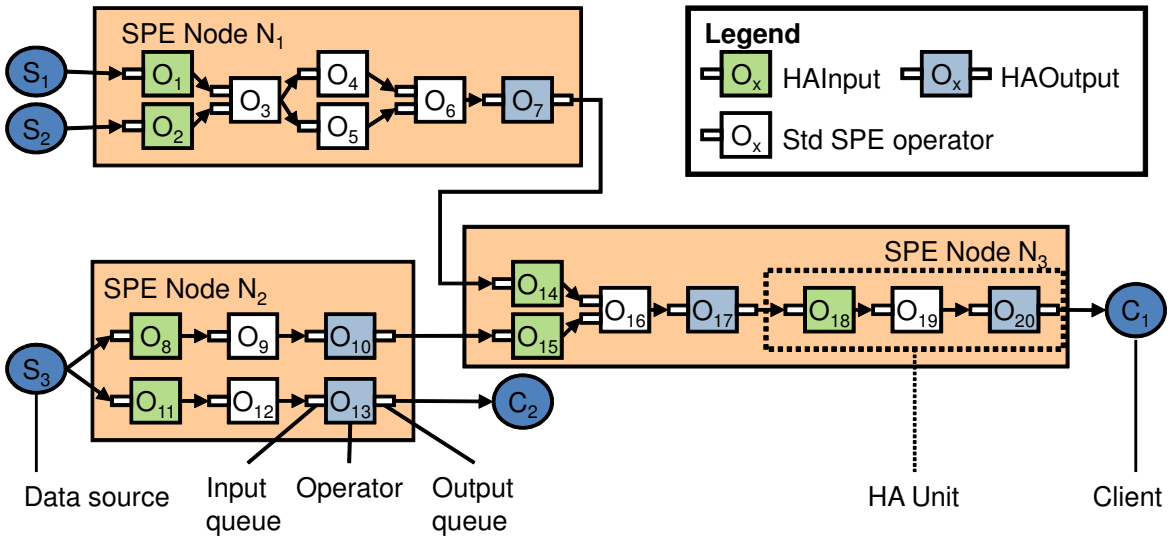


Figure 15: An SGuard query diagram. Operators O_1, O_2, O_8, O_{11}, O_{14}, O_{15} and O_{18} are HAInput operators; the remaining shaded boxes are HAOutput operators.

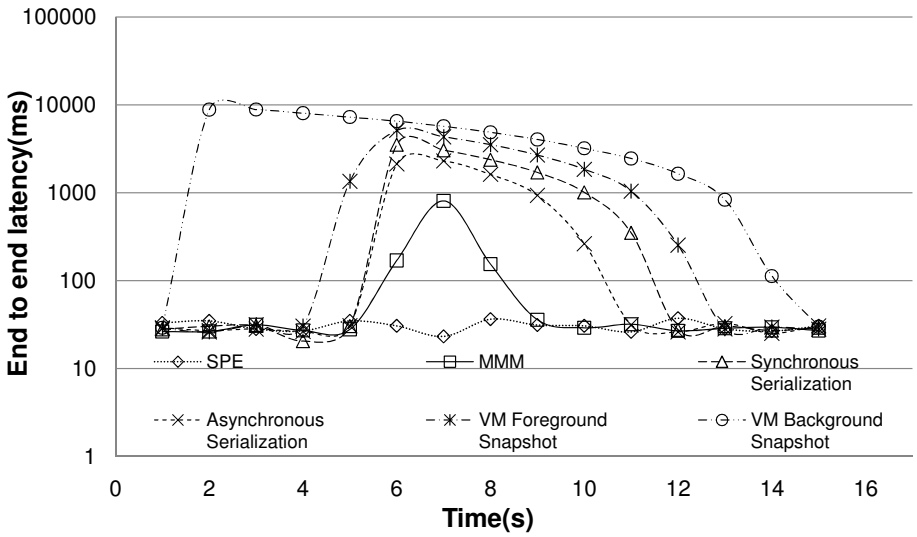
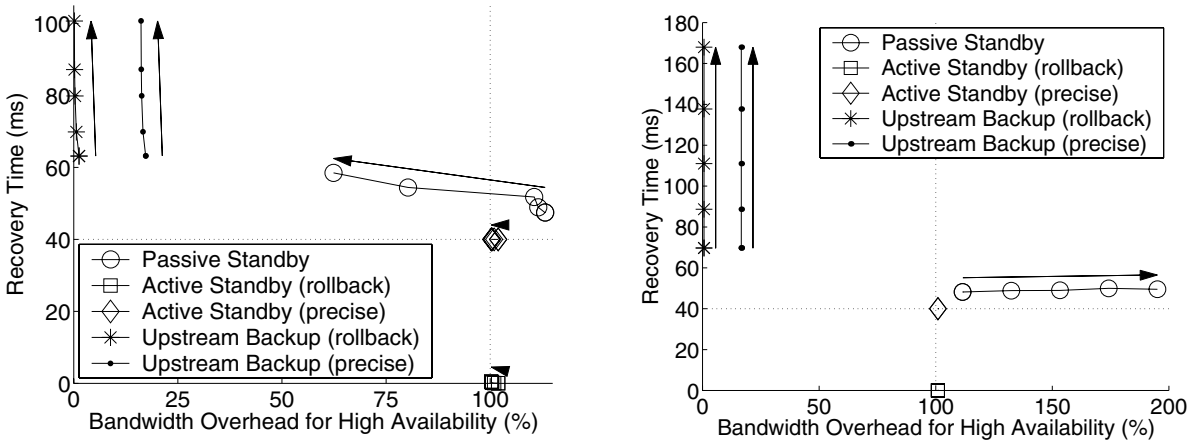


Figure 16: Query processing latencies for different SGuard checkpointing techniques. SPE does not checkpointing at all. Synchronous and asynchronous serialization both suspend processing while the checkpoint is serialized; the former write the checkpoint to disk before processing continues, the latter does not wait. VM foreground and background snapshots use a commercial virtual machine to serialize the entire process state while the processing is executing or while is it suspended, respectively.



(a) Effects of varying communication interval from 25 ms to 200 ms

(b) Effects of varying number of operators from 1 to 5

Figure 17: Comparison of simulated recovery techniques from Hwang et al. (2005). Arrows indicate change in parameter value.

ever, the flexibility and implementation benefits are clearly substantial. Figure 16 shows the effect on query processing latency of the various checkpointing methods during a checkpoint operation. The MMM technique clearly outperforms the others by a significant amount.

5 Comparison and Conclusions

In this survey, we have examined techniques to improve reliability in distributed query processing. For distributed query processing over replicated, static data, recovery techniques are an optimization; if queries are relatively short or if node failures are infrequent, it is always possible to simply restart the query with a stable set of nodes. Section 2.1 detailed the implementation of such a query processor, and described how it used upstream backup as a recovery technique to reuse incremental results and avoid redundant recomputation. This was shown to increase performance in their experiments. However, the experimental analysis was not terribly thorough, and certainly one can imagine cases (say, early on in a query's execution), when it is more efficient to simply restart a query. Given the lack of experimental results for other recovery methods in this context, it's hard to come to any general conclusions about recovery techniques for static data. Furthermore, some of the techniques for stream processing depended on idle time between tuples to perform fault-tolerance operations; it's not clear how these would work in a static data processing system, where the nodes may be CPU-bound.

In distributed stream processing engines, however, recovery techniques are needed to give correct, complete results in the event of a node failure. Data streams are typically read-once, and since operators may be stateful restarting a query (or failed operators) and continuing after a failure may never give the same results that would have been reported without failure. In stream engines, then, the decision is not whether to use one of the three recovery techniques we described, but which

one to use. A good starting point of any such discussion is the simulation results from Hwang et al. (2005), which are shown in Figure 17. Those results show that upstream backup has by far the least bandwidth overhead, but the highest recovery time; making it perhaps a good choice if failures are rare and the increased latency can be accommodated in the unlikely event of a failure. Active standby had the lowest recovery time, but used a significant amount of bandwidth and computational resources to do so. The results for passive standby were quite negative, giving it the worst aspects of active standby and upstream backup.

However, the more recent work in actual systems (as opposed to a simulation), detailed in Sections 4.1 and 4.2 has shown that, with a clever implementation, passive standby (i.e. checkpointing) can be a compelling technique that generally outperforms active standby. A combination of the techniques from those two lines of work might give an even larger advantage over active standby; however, it is not immediately obvious how well the two ideas would work together (though the MMM of Kwan et al. (2008) should be immediately applicable to the HA techniques of Hwang et al. (2007). These newer results lead to a revised conclusion, that favors active standby only when extremely low latency is required, even in the presence of failure; active standby may also require extra nodes, since half of the system resources are devoted to redundant computation. If failures are rare, and high latency can be tolerated while recovering from a failure, upstream backup is a good choice, since it has low impact on system performance or bandwidth use. If failures are more common, then passive standby gives a good compromise between bandwidth use, system capacity, and recovery time.

Reliability in distributed databases is a rich area, and one that has been studied in many contexts over the past thirty or so years. Recovery from failure during query processing has been an active area of research over the past decade, and while the papers surveyed here have made significant progress in improving the fault-tolerance of distributed stream processors and distributed databases in general, much work remains to be done. A more thorough experimental evaluation of the different recovery techniques in the same system would help to understand which techniques are best applied when, though the results so far can at least guide system developers in the right direction based on their general needs. Nevertheless, the fact that such systems are in active use outside the research community shows the power and versatility of the approaches developed so far, and should encourage those who continue to work in the area.

6 References

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stanley B. Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4):370–383, 2004.

Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1), 2008.

Joel F. Bartlett. A NonStop kernel. In *ACM Symposium on Operating Systems Principles*, pages 22–19, 1981.

Angelo R. Bobak. *Distributed and Multi-Database Systems*. Artech house, Boston, 1996.

Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226. Morgan Kaufmann, 2002.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Michael L. Scott and Larry L. Peterson, editors, *ACM Symposium on Operating Systems Principles*, pages 29–43. ACM, 2003. ISBN 1-58113-757-5.

Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In Hector Garcia-Molina and H. V. Jagadish, editors, *SIGMOD Conference*, pages 102–111. ACM Press, 1990.

Jim Gray. Why do computers stop and what can be done about it. Technial Report 85-7, Tandem Computers, Cupertino, CA, 1985.

Abdelsalam A. Helal, Abdelsalal A. Heddaya, and Bharat B. Bhargava. *Replication Techniques in Distributed Systems*. Kulwer Academic Publishers, Boston, 1996.

Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790. IEEE Computer Society, 2005. ISBN 0-7695-2285-8.

Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel, and Stanley B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, pages 176–185. IEEE, 2007.

YongChul Kwan, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. In *VLDB*, pages 574–585. ACM, 2008.

Bruce G. Lindsay. A retrospective of R*: A distributed database management system. *Proceedings of the IEEE*, 75(5):668–673, 1987.

M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, second edition edition, 1999.

James B. Rothnie, Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, T. A. Landers, Christopher L. Reeve, David W. Shipman, and Eugene Wong. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):1–17, 1980.

Mehul A. Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows*. PhD thesis, University of California, Berkeley, 2004.

Jim Smith and Paul Watson. Fault-tolerance in distributed query processing. In *International Database Engineering and Applications Symposium*, pages 329–338. IEEE Computer Society, 2005. ISBN 0-7695-2404-4.

Jim Smith, Paul Watson, Sandra de F. Mendes Sampaio, and Norman W. Paton. Polar: An architecture for a parallel ODMG compliant object database. In *Conference on Information and Knowledge Management*, pages 352–359. ACM, 2000.

Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. In Manish Parashar, editor, *GRID*, volume 2536 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 2002. ISBN 3-540-00133-6.

Acknowledgments

The author would like to thank the authors of the papers reviewed in this survey, both for making the effort to write thorough, thoughtful, and understandable discussions of their work, and for the figures, diagrams, and graphs which have been poached from their publications for reproduction here. He would also like to thank you, the reader, for persevering all the way to the end.