Fault-Tolerance in the Borealis Distributed Stream Processing System

MAGDALENA BALAZINSKA
University of Washington
and
HARI BALAKRISHNAN, SAMUEL R. MADDEN, and MICHAEL STONEBRAKER
Massachusetts Institute of Technology

Over the past few years, Stream Processing Engines (SPEs) have emerged as a new class of software systems, enabling low latency processing of streams of data arriving at high rates. As SPEs mature and get used in monitoring applications that must continuously run (*e.g.*, in network security monitoring), a significant challenge arises: SPEs must be able to handle various software and hardware faults that occur, masking them to provide high availability (HA). In this paper, we develop, implement, and evaluate DPC (Delay, Process, and Correct), a protocol to handle crash failures of processing nodes and network failures in a distributed SPE.

Like previous approaches to HA, DPC uses replication and masks many types of node and network failures. In the presence of network partitions, the designer of any replication system faces a choice between providing availability or data consistency across the replicas. In DPC, this choice is made explicit: the user specifies an availability bound (no result should be delayed by more than a specified delay threshold even under failure if the corresponding input is available), and DPC attempts to minimize the resulting inconsistency between replicas (not all of which might have seen the input data) while meeting the given delay threshold. Although conceptually simple, the DPC protocol tolerates the occurrence of multiple simultaneous failures as well as any further failures that occur during recovery.

This paper describes DPC and its implementation in the Borealis SPE. We show that DPC enables a distributed SPE to maintain low-latency processing at all times, while also achieving eventual consistency, where applications eventually receive the complete and correct output streams. Furthermore, we show that, independent of system size and failure location, it is possible to handle failures almost up-to the user-specified bound in a manner that meets the required availability without introducing any inconsistency.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems — Distributed databases; D.4.5 [**Operating Systems**]: Reliability – Fault-tolerance

General Terms: Algorithms, Design, Experimentation, Reliability

Additional Keywords and Phrases: Distributed stream processing, fault-tolerance, availability, consistency

Authors' addresses: M. Balazinska, 550 Paul G. Allen Center. Dept. of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA, 98195; email: magda@cs.washington.edu; S. R. Madden, H. Balakrishnan, and M. Stonebraker, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139; email: {madden,hari,stonebraker}@csail.mit.edu;

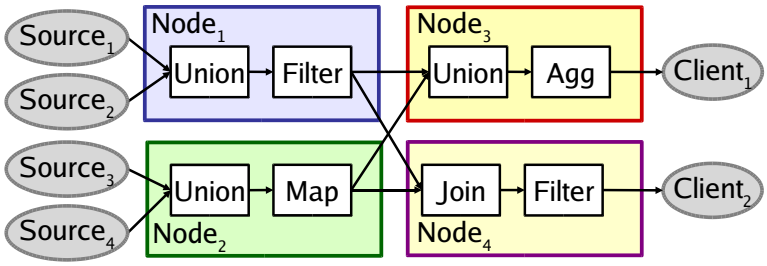


Fig. 1. **Query diagram in a distributed SPE.** Node_1 and Node_2 are upstream neighbors of Node_3 and Node_4.

1. INTRODUCTION

Monitoring applications, such as network intrusion detection, computer-system monitoring, financial services, sensor-based environment monitoring (*e.g.*, highway monitoring, pipeline health monitoring), or military applications (*e.g.*, platoon tracking), require continuous processing of information from geographically distributed data sources: *e.g.*, sensors, network monitors, computer system monitors, ticker feeds. These applications require results to be produced at low latency even in the face of high and variable input data rates or system failures.

Stream processing engines (SPEs) [Abadi et al. 2005][StreamBase] (also known as stream databases [Cranor et al. 2003], data stream managers [Abadi et al. 2003; Motwani et al. 2003], continuous query processors [Chandrasekaran et al. 2003], complex event processing engines [Coral8], or event stream processors [Aleri]) are a class of software systems that handle the data processing requirements of monitoring applications. In these systems, the application-logic takes the form of a dataflow composed of a relatively small set of operators (*e.g.*, filters, aggregates, and correlations). To avoid blocking in face of unbounded data inputs, these operators typically perform their computations on windows of data that move with time. For example, an aggregate operator may produce the average room temperature every minute. Many SPEs allow operators to be spread across multiple processing nodes [Abadi et al. 2005; Cherniack et al. 2003; Shah et al. 2004]. In such distributed deployments, each SPE node produces result streams that are either sent to applications or to other nodes for additional processing. Figure 1 illustrates a distributed SPE performing a computation spread across four nodes. When a stream goes from one node to another, the nodes are called *upstream and downstream neighbors*.

When a processing node fails or becomes disconnected, a distributed SPE may block or it may produce erroneous results. The likelihood of a failure increases with the number of processing nodes in the system. Ideally, the distributed SPE should mask any internal failures from users, ensuring that they receive correct results (*i.e.*, maintaining consistency) and that they receive these results quickly (*i.e.*, maintaining low processing latency, a form of availability). In this paper, we

investigate techniques to achieve such fault-tolerant distributed stream processing.

The traditional approach to masking failures is through replication [Gray et al. 1996], running multiple copies of each operator on distinct processing nodes. With replication, if a processing node fails or becomes disconnected the system can continue operating correctly by using a replica of the failed node [Hwang et al. 2005; Shah et al. 2004]. Replication cannot mask all system failures, though. If a network partition prevents a node from communicating with *all* replicas of one of its upstream neighbors, the node must either continue processing the remaining data to maintain availability or block to maintain consistency. Previous techniques for fault-tolerant stream processing either do not address network failures [Hwang et al. 2005] or strictly favor consistency over availability when system partitions occur [Shah et al. 2004]. In contrast, we propose a fault-tolerance scheme, called *Delay, Process, and Correct* (DPC) that not only handles both node and network failures but also accommodates applications with different desired trade-offs between availability and consistency in face of network partitions.

The key idea of DPC is to let each application define its desired trade-off between availability and consistency by specifying the *maximum incremental processing latency* it can tolerate. DPC maintains this application-specific processing latency at all times, independent of failure durations. DPC achieves this goal by favoring availability over consistency when long partitions occur: *i.e.*, each SPE node guarantees that it will process all available inputs within a predefined time period even if other inputs are unavailable. The best-effort data produced in this manner is called *tentative*. During failures, we measure the level of inconsistency of a replica by counting the number of tentative result tuples it produces. The goal of DPC is to minimize inconsistency by minimizing the number of tentative tuples, while maintaining the required latency threshold. Such functionality is important for many applications. For example, in a distributed network monitoring system, if some monitoring nodes become unreachable, continuing to process data from the remaining nodes can help detect at least a subset of all anomalous conditions. In this application, it is acceptable to delay processing data for a few seconds but not much longer than this, if doing so will produce fewer false positive and false negative alerts. As another example, consider a sensor-based environment monitoring application (*e.g.*, pipeline health monitoring or building air-quality monitoring). In this application, if a subset of sensors fails or becomes disconnected, the system may need to produce alerts tentatively as the alerts are based only on partial information. Technicians may then be dispatched to make final diagnostics. In this domain, applications may be able to wait for tens of seconds or even a couple of minutes if doing so will produce more accurate results. In general, DPC gives applications the choice: an application that needs to see results quickly, even if these results are not accurate, can specify a low latency threshold. An application that cannot handle inconsistency can set an infinite threshold.

At the same time, DPC ensures eventual consistency: even though failures occur, as long as they also heal, clients eventually receive the complete and correct output stream (*i.e.*, all tentative tuples are corrected). In the network monitoring scenario, this means that the administrator eventually sees the complete list of problems that occurred during the partition. Such functionality is also important:

the network administrator eventually needs to know the exact list of all potential security breaches and infected machines because they may require further action such as cleaning and patching the machines. Similarly, in the sensor-based environment monitoring application, once failures heal the system can update the status of previously produced alerts to either real alerts or false alarms. If the system can do so soon after the failure heals, technicians dispatched to fix raised problems can be quickly re-assigned as needed. Hence, in DPC, all applications eventually see correct results, independent of their specific availability and consistency constraints.

In this paper, we present and evaluate DPC.[1] In Section 2, we first outline necessary assumptions and discuss the goals that a fault-tolerance mechanism for a distributed SPE should achieve. We present the basic DPC scheme in Sections 3 and 4. DPC is designed, implemented, and evaluated in Borealis [Abadi et al. 2005], but the basic underlying techniques and principles are more generally applicable.

In Section 5, we present experimental results showing that DPC can ensure eventual consistency while continuously maintaining availability. For example, DPC successfully maintains processing latency within a required 3-second bound even during a 60-second failure and subsequent recovery, during which earlier results are corrected while new data continues to be processed.

In Section 6, we investigate techniques to reduce inconsistency by leveraging the application-defined tolerance to bounded increases in processing latency. We show that delaying tuples as much as possible always reduces inconsistency for a single SPE node. For a chain of nodes, we demonstrate that, contrary to intuition, delaying improves consistency only for short-duration failures. Furthermore, we show that, in order to minimize inconsistency, the system should not simply divide the incremental latency across nodes. Instead, when a node first detects a failure, it should always block for the maximum tolerable incremental latency (minus queuing delays), independently of its location in the distributed system. With this approach, we show an example where the incremental processing latency is 8 seconds, but even a system of four processing nodes in sequence can mask failures up to 6.5 seconds in duration.

We investigate the overhead of DPC in Section 7. In Section 8, we discuss how failures can be isolated from each other such that, independently of the assignment of operators to processing nodes, only those operators affected by a failure experience any delays and undergo any recovery. We also discuss the details of buffer management and long-duration failures: for a large class of queries, DPC can continue to maintain availability even after buffers fill-up, yet still guarantee that the system will converge back to a consistent state and that a *predefined window of most recent results* will be corrected after the failure heals. Finally, we review related work in Section 9 before concluding in Section 10.

[1]This paper is an extended version of [Balazinska et al. 2005]. In this paper, we present a more detailed version of the approach, a deeper study of availability and consistency trade-offs in *distributed* deployments, and techniques for allocating delays to processing nodes. We omit the comparative study of state reconciliation techniques with either undo/redo or checkpoint/redo, as we demonstrated in [Balazinska et al. 2005] that the latter technique outperforms the former.

2. MODEL, ASSUMPTIONS, AND GOALS

In this section, we describe our distributed stream processing model, failure assumptions, and design goals.

2.1 Query Model

In Borealis [Abadi et al. 2005], a query takes the form of a loop-free, directed graph of operators. Each operator processes data arriving on its input streams and produces data on its output stream. These graphs are called *query diagrams*. Figure 1 illustrates a query diagram distributed across four nodes. Borealis operators come from Aurora [Abadi et al. 2003]. In this paper, we only consider the following fundamental operators [Abadi et al. 2003].

(1) Filter: tests each input tuple against a predicate.
(2) Map: transforms each input tuple into a single output tuple.
(3) Aggregate: computes aggregate functions over windows of data that slide with time (possibly grouping the data first).
(4) Join: joins tuples on streams when these tuples fall within some time window of each other.
(5) Union: merges tuples from two or more input streams into a single output stream.

In particular, we ignore Read and Write operators [Balakrishnan et al. 2004], which store data in persistent storage, as these operators are not fundamental to stream processing.

In general, we restrict DPC to deterministic operators. We consider an operator to be *deterministic* if its results do not depend on the times at which its inputs arrive (*e.g.*, the operator does not use timeouts nor randomization); of course, the results will usually depend on the input data order, *including the inter-arrival order of tuples on the different input streams*. Based on this definition, all the above Borealis operators (Filter, Map, Aggregate, Join, and Union) are deterministic (aggregate requires setting its `independent-window-alignment` parameter to ensure the window boundaries are independent of the exact value of the first processed tuple). We further discuss the implications of deterministic operators in Section 4.2.

To avoid blocking in the face of infinite input streams, operators perform their computations over windows of tuples. For example, an aggregate operator may compute the average temperature *every hour*. An operator may join temperature readings with light readings when the two are taken at the same location and *within five seconds of each other*. Some operators, such as Join, still block when some of their input streams are missing. In contrast, a Union is an example of a *non-blocking* operator because it can perform meaningful processing even when some of its input streams are missing. In Figure 1, the failure of a data source does not prevent the system from processing the remaining streams. Failure of Node_1 or Node_2 does not block Node_3 but blocks Node_4.

DPC achieves fault-tolerance through replication; each operator in the query diagram is instantiated on at least two distinct processing nodes. All replicas process input streams and produce output streams enabling a downstream node to get its input streams from any replica of each upstream neighbor. Figure 2 illustrates a replicated version of the query diagram from Figure 1. The figure

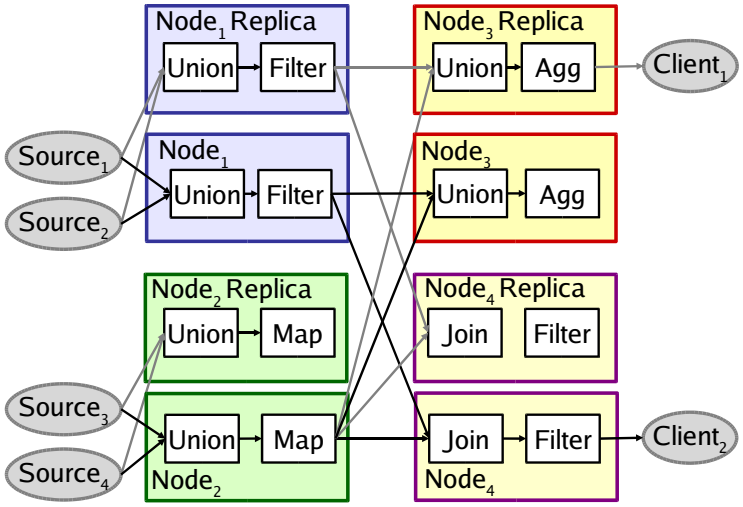


Fig. 2. **Distributed and replicated query diagram.** The diagram shows one possible data flow.

shows one possible data flow. At any point in time, outputs produced by some nodes may be unused.

2.2 Assumptions and Failure Model

We assume that data sources have loosely synchronized clocks and can timestamp the tuples they push into the system. When two or more streams are joined, unioned, or otherwise combined by an operator, DPC delays tuples until timestamps match on all streams. The clocks at data sources must therefore be sufficiently synchronized to ensure these buffering delays are smaller than the maximum incremental processing latency, X, specified by the application. We make the same assumption about the timestamps that operators assign to their output tuples. In Borealis, these timestamps are computed from the input-tuple timestamps. Application-defined attributes serving in window specifications have analogous requirements; Borealis applications already use such attributes.

We further assume that each processing node has sufficient resources (CPU, memory, and network bandwidth) to keep up with tuple input rates ensuring that queues do not form in the absence of failures. We assume that the network latency between any pair of nodes is small compared with the maximum incremental processing latency, X.

We consider dynamic changes to the diagram or its deployment (*i.e.*, the assignment of operators to processing nodes) outside of the scope of this paper. We thus assume that the query diagram, its deployment, and the set of replicas for each processing node are static.

Finally, we assume that data sources and clients implement DPC. This can be achieved by having clients and data sources use a fault-tolerant library or by having them communicate with the system through proxies (or nearby processing nodes)

that implement the required functionality. We also assume that data sources, or proxies acting on their behalf, log input tuples persistently (*e.g.*, in a transactional queue [Bernstein et al. 1990]) before transmitting them to all replicas that process the corresponding streams. A persistent log ensures that all replicas eventually see the same input tuples, in spite of proxy or data source failures. The fail-stop failure of a data source, however, causes the permanent loss of input tuples that would have otherwise been produced by the data source.

As we present DPC, we begin by assuming that all tuples ever produced by a data source or a processing node are buffered. We revisit this assumption in Section 8.1, where we discuss buffer management and long-duration failures. Except for data sources, we assume that buffers are lost when a processing node fails.

With the assumption that tuples are logged forever and that data sources persistently log the data they produce, DPC can cope with the crash failure of all processing nodes. During such a failure, clients may not receive any data. After failed nodes recover, they can reprocess all tuples logged upstream ensuring eventual consistency. If buffers are truncated, DPC handles the simultaneous crash failure of at most $R - 1$ of the R replicas of each node. At any time, at least one replica of each node must hold the current consistent state. This state comprises the set of input tuples no longer buffered upstream and the set of output tuples not yet received by all replicas of all downstream neighbors. DPC also handles network failures that cause message losses and delays, preventing any subset of nodes from communicating with one another, possibly partitioning the system. DPC can handle multiple failures overlapping in time.

For simplicity, we assume that replicas communicate using a reliable, in-order protocol like TCP. With this assumption, because tuples are never re-ordered on a stream, a downstream node can indicate with a single tuple identifier the exact data it has received so far.

DPC achieves fault-tolerance by replicating all operators on multiple processing nodes. Such replication, of course, imposes a significant overhead. With our approach, the bulk of the overhead grows linearly with the number of replicas: for the same input load, a fault-tolerant system with N replicas per processing node requires N times the amount of hardware. Conversely, given N machines, a replicated system can process approximately 1/Nth of the load of a non-replicated system that would spread its load across all N machines. Because DPC is designed to operate on high-end servers deployed on the Internet, we believe such overhead is acceptable for a low level of replication (*e.g.*, two or three replicas per processing node).

In the setting of servers deployed on the Internet, we also expect failures to occur with low frequency. DPC supports multiple concurrent failures, but it is designed for a low failure frequency, where processing nodes typically have time to recover between consecutive failures.

2.3 Design Goals

In this section, we discuss the goals that a fault-tolerance mechanism for a distributed stream processing system should achieve, and present the resulting desired properties of DPC.

2.3.1 *Availability Goal.* Because many stream processing applications are geared toward monitoring tasks, when a failure occurs upstream from a non-blocking operator and causes some (but not all) of its input streams to be unavailable, it is often useful to continue processing the inputs that remain available. For example, in the network monitoring application, even if only a subset of monitors are available, processing their data might suffice to identify some potential attackers or other network anomalies. In this application, low latency processing is critical to mitigate attacks. However, some events might go undetected because a subset of the information is missing and some aggregate results may be incorrect. Furthermore, the state of replicas diverges as they process different inputs.

The traditional definition of availability requires only that the system eventually produces a response for each request [Gilbert and Lynch 2002]. Availability may also measure the fraction of time that the system is operational and servicing requests [Gray and Reuters 1993]. In an SPE, however, because client applications passively wait to receive output results, we define availability in terms of processing latency, where a low processing latency indicates a high level of availability.

To simplify our problem, we measure availability in terms of *incremental processing latency*. When an application submits a query to the system, DPC allows the application to specify a desired availability, X, as a maximum incremental processing latency that the application can tolerate on its output streams (the same threshold applies to all output streams within the query). For example, in a query diagram that takes 200 milliseconds to transform a set of input tuples into an output result, a client can request "no more than 100 milliseconds of added delay", and DPC should ensure that output results are produced within 300 milliseconds.

With the above definition, to determine if the system meets a given availability requirement, we only need to measure the *extra* buffering and delaying imposed on top of normal processing. We define $\texttt{Delay}_{\texttt{new}}$ as the maximum *incremental* processing latency for *any* output tuple and express the availability goal as $\texttt{Delay}_{\texttt{new}} < X$. With this definition, we express the main goal of DPC as:

PROPERTY 1. *DPC ensures that as long as some path of non-blocking operators is available between one or more data sources and a client application, the client receives results within the desired availability requirement: the system ensures that* $\texttt{Delay}_{\texttt{new}} < X$.

As we discuss later, DPC divides X between processing nodes. To ensure Property 1, a node that experiences a failure on an input stream must switch to another replica of its upstream neighbor, if such a replica exists, within D time-units of arrival of the oldest unprocessed input tuples. If no replica exists, the node must process all tuples that are still available, within D time-units of their arrival, where D is the maximum incremental processing latency assigned to the node.

Even though we only measure incremental latencies, we can show how $\texttt{Delay}_{\texttt{new}}$ relates to normal processing latency. We define $\texttt{proc}(u)$ as the normal processing latency of an *output* tuple, u, in the absence of failure. $\texttt{proc}(u)$ is the difference between the time when the SPE produces u and the time when the oldest input tuple that contributed to the value of u entered the SPE. Given $\texttt{proc}(u)$ and the actual processing latency of a tuple, $\texttt{delay}(u)$, $\texttt{Delay}_{\texttt{new}} = \max_{u \in \texttt{NewOutput}}(\texttt{delay}(u) -$

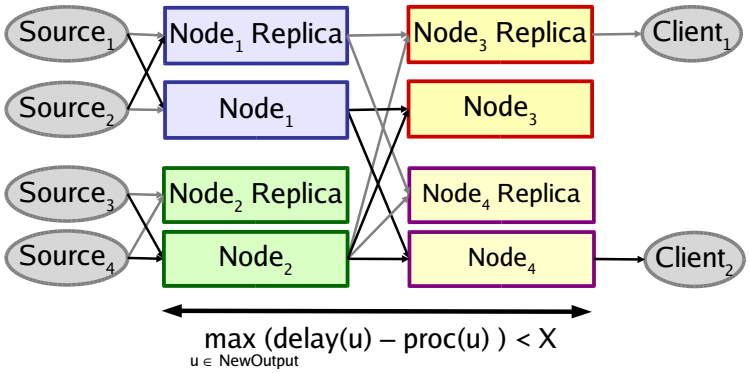


Fig. 3. **Availability guarantee.**

$\texttt{proc}(u)$), where `NewOutput` is the set of output tuples produced by the query diagram. Figure 3 illustrates the availability guarantee of DPC.

2.3.2 *Consistency Goal.* Even when they favor availability, many stream processing applications need to receive the correct results eventually. For example, in the network monitoring application, the administrator eventually needs to see the complete set of intrusions or failures because these intrusions and failures may require additional actions such as cleaning and patching a compromised host. Our second goal is thus for DPC to provide eventual consistency.

In traditional optimistic replication schemes, eventual consistency requires that all replicas of an object eventually process all update operations in an equivalent order (*i.e.*, the prefix of operations in the final order must grow monotonically over time at all replicas [Saito and Shapiro 2005]). Replicas can, however, temporarily process updates in different orders to provide high availability.

In an SPE, the state of processing nodes is transient and the output stream continuous. We thus translate eventual consistency as requiring that all replicas of the same query diagram fragment eventually process the same input tuples in the same order, and that order should be one that could have been possible at a processing node without failure.

Eventual consistency is a property of a replicated object. With the traditional notion of eventual consistency, responses to operations performed on the object do not have to be corrected after operations are reprocessed in their final order [Fekete et al. 1996]. In an SPE, because the output of processing nodes serves as input to their downstream neighbors, we extend the notion of eventual consistency to include output streams. We require that each replica eventually processes the same input tuples in the same order and *produces the same output tuples in the same order*.

In summary, the second goal of DPC is:

PROPERTY 2. *Assuming sufficiently large buffers, ensure* eventual consistency.

We define *eventual consistency* as follows:

DEFINITION 1. *A replicated SPE maintains* eventual consistency *if all replicas of the same query diagram fragment eventually process the same input tuples in the*

same order and produce the same output tuples in the same order, and that order could have been produced by a single processing node without failure.

Throughout the paper, we assume that nodes have enough space to buffer all tuples that accumulate during failures. We discuss buffer management in Section 8.1.

2.3.3 *Measuring Inconsistency.* In traditional optimistic replication schemes, once the final processing order of some operations is known, the operations are said to be *stable* [Fekete et al. 1996]. We use the same definition for tuples. An input tuple is stable once its final processing order is known. When a replica processes stable input tuples, it produces stable output tuples because these output tuples have final values and appear in final order. Eventual consistency ensures that clients eventually receive stable versions of all results.

All intermediate results that are produced in order to provide availability, and are not stable, are called *tentative*. At any point in time, as a measure of inconsistency, we use $\texttt{N}_{\texttt{tentative}}$, the *number of tentative tuples produced on all output streams* of a query diagram. $\texttt{N}_{\texttt{tentative}}$ may also be thought of as a (crude) substitute for the degree of divergence between replicas of the same query diagram when the set of input streams is not the same at the replicas. More specifically, we use the following definition:

DEFINITION 2. *The* inconsistency *of a stream s, $\texttt{N}_{\texttt{tentative}}(s)$, is the number of tentative tuples produced on s since the last stable tuple. The inconsistency, $\texttt{N}_{\texttt{tentative}}$, of a query diagram is the sum of $\texttt{N}_{\texttt{tentative}}(s)$ for all output streams, s.*

Note that $\texttt{Delay}_{\texttt{new}}$ only measures the availability of result tuples that carry new information. `NewOutput` does not include any stable results that correct previously tentative ones.

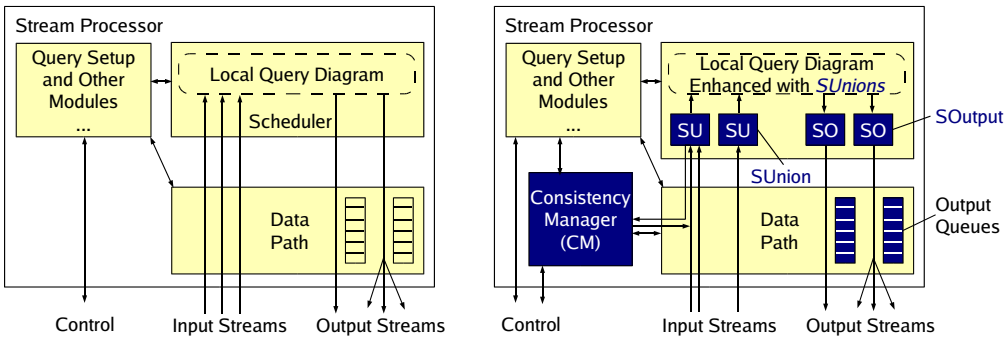
2.3.4 *Minimizing Inconsistency Goal.* The main goal of DPC is thus to ensure that the system meets, if possible, a pre-defined availability level while ensuring eventual consistency. To maintain availability, the system may produce tentative tuples. To ensure eventual consistency, tentative tuples are later corrected with stable ones. Because it is expensive to correct earlier results in an SPE, we seek to minimize the number of tentative tuples. In the absence of failures, we would like replicas to remain mutually consistent ensuring that all results are stable. If a failure occurs, we would like the system to mask the failure without introducing inconsistency if possible. Finally, if a failure cannot be masked, we would like the system to minimize the number of tentative results. We summarize these requirements with the following two properties that we would like DPC to provide:

PROPERTY 3. *DPC favors stable results over tentative results when both are available.*

PROPERTY 4. *Among possible ways to achieve Properties 1 and 2, we seek methods that minimize $\texttt{N}_{\texttt{tentative}}$.*

3. EXTENDED SPE SOFTWARE ARCHITECTURE

Achieving the fault-tolerance goals described in the previous section requires changes to the software architecture of an SPE. In this section, we present an



(a) Original architecture

(b) Extended architecture

Fig. 4. **SPE software architecture extensions to support DPC.** Arrows indicate communication between components.

overview of the high-level changes required by DPC. We discuss these changes further as we present the details of the approach in the following section.

As illustrated in Figure 4(a), an SPE typically comprises at least three different logical components: a component to setup and modify the locally running query diagram, a component to continuously execute the local query diagram, and a component, which we call *Data Path*, to manage the data entering and exiting the node.

As illustrated in Figure 4(b), to enable fault-tolerance, we propose the following set of changes to these components:

(1) **Inter-node communication**: Fault-tolerance requires communication between upstream and downstream nodes and node replicas. We introduce a new component, the *Consistency Manager* to carry-out these runtime communications and modify connections between nodes as failures occur and heal.

(2) **Intra-node state monitoring**: In DPC, a processing node can be in different consistency states that it must communicate to other nodes. It must also reconcile its state after failures heal. Because of its global role, the Consistency Manager performs these functions: it determines the current consistency state of the local node, it advertises that state to other nodes, and it decides when to reconcile the state of the node after a failure heals.

(3) **Stream stabilization**: In response to failures, nodes switch upstream neighbors. To ensure that such switches do not disrupt stream processing, we extend the Data Path with extra input stream monitoring and output stream buffering and replaying capabilities. These capabilities enable the Data Path to control the data entering a node from upstream and ensure new downstream neighbors continue receiving data from the correct point in the stream.

(4) **Query diagram extensions**. The query diagram requires three changes. First, it must be made deterministic such that multiple replicas of the same query diagram fragment remain consistent in the absence of failures (note that even when composed of only deterministic operators, a query diagram may not be deterministic itself). We enable this feature by augmenting the local query

diagram with a new operator: *SUnion*. SUnion operators are inserted in front of all operators with more than one input streams. They ensure that replicas of these operators process tuples in the same order. In Borealis, only two operators have more than one input stream: Union and Join. SUnion replaces the Union operator and is also placed in front of all Join operators. The Join operator is also slightly modified to always process input tuples in the order prepared by the preceding SUnion. We call the modified Join operator SJoin. To ensure consistency, SUnion operators delay tuples on different streams until their timestamps match. To avoid delaying tuples unnecessarily when data rates are low, we introduce the notion of `boundary` tuples, which act both as punctuation tuples and heartbeats.

SUnion operators ensure replicas remain consistent in the absence of failures and, when failures occur, they also control the trade-offs between availability and consistency by delaying processing tuples as appropriate. For this reason, we also insert SUnion operators on all input streams to the processing node. This enables SUnions to delay any tentative input tuples as appropriate.

Second, the query diagram must support the notion of stable, tentative, and boundary tuples. To achieve this goal, all operators, including SUnion, are modified to correctly label their output tuples as either stable or tentative based on the type of the input tuples they process. They are also modified to correctly process boundary tuples. In our implementation, in addition to implementing SUnion, we modified the following Borealis operators: Filter, Map, Aggregate, and SJoin.

Third, to ensure eventual consistency, DPC requires the ability to rollback a query diagram to an earlier state and reprocess all input data from there. To enable this functionality, all operators are extended with the ability to save and recover their state from a checkpoint. We also introduce a new operator, *SOutput*, that drops all duplicate tuples produced during this reprocessing and signals the end of reconciliation to the Consistency Manager.

4. BASIC APPROACH

In this section, we present the basic DPC approach and algorithms. In DPC, each replica manages its own availability and consistency by implementing the state machine shown in Figure 5 that has three states: STABLE, UPSTREAM_FAILURE (UP_FAILURE), and STABILIZATION.

As long as all upstream neighbors of a node are producing stable tuples, the node is in the STABLE state (in Section 4.2, we discuss how DPC handles sporadic tuple production by upstream neighbors). In this state, the node processes tuples as they arrive and passes stable results to downstream neighbors. To maintain consistency between replicas that may receive inputs in different orders, we define a data-serializing operator, *SUnion*. Section 4.2 discusses the STABLE state and the SUnion operator.

If one input stream becomes unavailable or starts carrying tentative tuples, a node goes into the UP_FAILURE state. In that state, the node tries to find another stable source for the input stream. In no stable source is available, to maintain low processing latency, the node may have to continue processing the inputs that

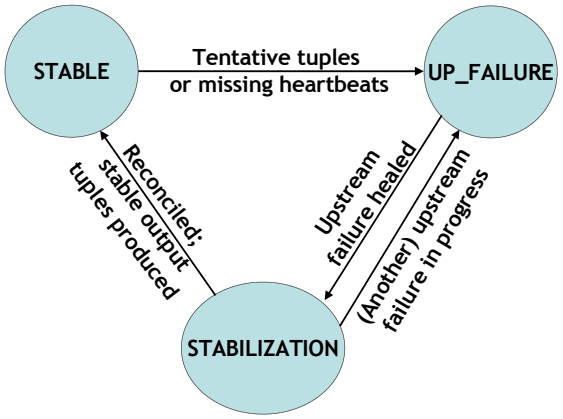


Fig. 5. **The DPC state machine.**

remain available. The node may, however, delay processing these tuples if doing so improves consistency. Section 4.3 discusses the UP_FAILURE state.

A failure *heals* when a previously unavailable upstream neighbor starts producing stable tuples again or when a node finds another replica of the upstream neighbor that can provide the stable version of the stream. Once a node receives the stable versions of all previously missing or tentative input tuples, it transitions into the STABILIZATION state. In this state, if the node processed any tentative tuples during UP_FAILURE it must now reconcile its state and stabilize its outputs. We present the STABILIZATION state in Section 4.4.

During STABILIZATION, new input tuples are likely to continue to arrive. To maintain the required low processing latency, the node must ensure these new tuples are processed within the required time-bound in spite of the ongoing state reconciliation. Our approach enables a node to reconcile its state and correct its outputs, while ensuring that new tuples continue to be processed as we discuss in Section 4.4.

Once stabilization completes, the node transitions to the STABLE state if there are no other current failures, or back to the UP_FAILURE state otherwise.

4.1 Data Model

With DPC, nodes and applications must distinguish between stable and tentative results. Stable tuples produced after stabilization may override previous tentative ones, requiring a node to correctly process these amendments. Traditionally, a stream is an append-only sequence of tuples of the form: $(t, a_1, \ldots, a_m)$, where t is a timestamp value and $a_1, \ldots, a_m$ are attribute values [Abadi et al. 2003]. To accommodate our new tuple semantics, we extend the Borealis data model [Abadi et al. 2005] as follows. In Borealis, tuples take the following form (we ignore header fields that DPC does not use):

$$(\text{tuple\_type}, \text{tuple\_id}, \text{tuple\_stime}, a_1, \ldots, a_m)$$

(1) **tuple_type** indicates the type of the tuple.

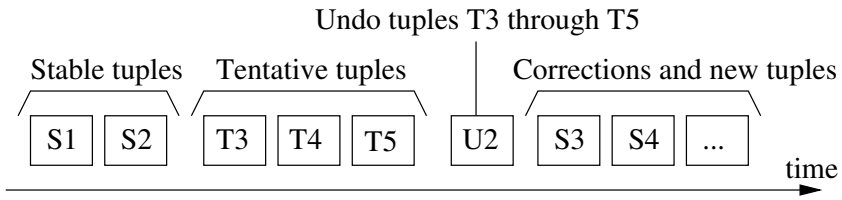


Fig. 6. **Example of using tentative and undo tuples. U2 indicates that all tuples following tuple with** tuple_id **2 (S2 in this case) should be undone.**

(2) **tuple_id** uniquely identifies the tuple in the stream.
(3) **tuple_stime** is a new tuple timestamp.[2]

Traditionally, all tuples are immutable stable insertions. We introduce two new types of tuples: TENTATIVE and UNDO. A tentative tuple is one that *results from processing a subset of inputs and may subsequently be amended with a stable version.* For example, a Union operator that loses an input stream can process tuples from the remaining live streams. In that case, the Union should label its output tuples as tentative. All operators downstream from the Union that process these tentative tuples should label their output tuples as tentative as well. Unlike a Union, a Join operator with a missing input stream will block and will not produce any tuples during a failure. However, a Join operator that processes tentative tuples also produces tentative tuples.

An undo tuple indicates that a suffix of tuples on a stream should be deleted and the associated state of any operators rolled back. As illustrated in Figure 6, the undo tuple indicates the suffix with the tuple_id of the last tuple not to be undone. Stable tuples that follow an undo replace the undone tentative tuples. Applications that do not tolerate inconsistency may thus simply wait to receive stable tuples. Table I summarizes the new tuple types. As shown in the table, we use a few additional tuple types in our approach but they do not fundamentally change the data model. In particular, SUnion and SOutput operators use separate control output streams to communicate their state to the Consistency Manager. The latter uses this information to determine the overall state of the node and take actions: the Consistency Manager updates and advertises the correct overall state to other nodes; when all previously failed SUnions have received some corrected tuples on their input streams, the Consistency Manager can trigger state reconciliation.

4.2 STABLE State

To minimize inconsistency and facilitate failure handling, DPC ensures that all replicas remain mutually consistent in the absence of failures: that they process the same input in the same order, go through the same internal computational states, and produce the same output in the same order. In this section, we first present how DPC ensures mutual replica consistency. In the STABLE state, nodes must also detect failures of their input streams in order to transition to the UP_FAILURE state. We discuss failure detection second.

[2]In Borealis, tuples also have a *separate* timestamp field used for quality of service purposes.

Tuple type	Description
Data streams	
INSERTION	Regular stable tuple.
TENTATIVE	Tuple that results from processing a subset of inputs and may later be corrected.
BOUNDARY	All following tuples will have a timestamp equal or greater to the one indicated.
UNDO	Suffix of tuples should be rolled back.
REC_DONE	Tuple that indicates the end of reconciliation.
Control streams	Signals from SUnion or SOutput to the Consistency Manager
UP_FAILURE	Entering inconsistent state (SUnion).
REC_REQUEST	Input was corrected, can reconcile state (SUnion).
REC_DONE	Same as above (SOutput).

Table I. **New tuple types.**

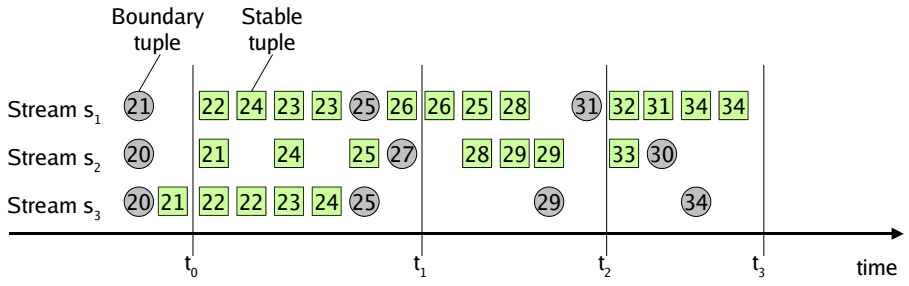
4.2.1 *Serializing Input Tuples.* As we mentioned above, we consider only deterministic operators. If all operators are deterministic, we only need to ensure that replicas of the same operator process data in the same order to maintain consistency; otherwise, the replicas will diverge even without failures. Since we assume that nodes communicate using a reliable in-order protocol like TCP, tuples never get re-ordered within a stream. To ensure consistency, however, we still need a way to order tuples deterministically *across multiple input streams that feed the same operator* (*e.g.*, Union and Join). Indeed, two replicas of an operator with two input streams A and B may receive tuples in a different interleaved order: one replica, located closer to the source of A, may receive tuples on stream A before tuples on stream B. The other replica, closer to the source of B, may receive tuples on that stream first.

Because tuples on streams are not necessarily sorted on any attribute and they may arrive at significantly different rates, we propose to compute a total order by inserting additional *boundary tuples* into streams. Boundary tuples have tuple_type = BOUNDARY and serve the role of both punctuation tuples [Tucker and Maier 2003] and heartbeats [Srivastava and Widom 2004]. The punctuation property of boundary tuples requires that no tuples with tuple_stime smaller than the boundary's tuple_stime appear after the boundary on the stream. Hence, an operator with i input streams can deterministically order all tuples that satisfy:

$$\texttt{tuple\_stime} < \min_{\forall i}(b_i), \tag{1}$$

where b_i is the tuple_stime value of the latest boundary tuple received on stream i. Note that tuples in a stream appearing between two boundaries can be out-of-order with respect to their timestamps. Figure 7 illustrates the approach for three streams. In the example, at time t_0, $\min(b_i) = 20$, and all tuples with tuple_stime values strictly below 20 can be deterministically ordered. Similarly at time t_1, tuples below 25 can be ordered. At t_2, only tuples below 27 can be ordered, since the last boundary tuple on stream S_2 had value 27. Finally, at t_3 all tuples up to 30 can be ordered.

Rather than modifying operators to sort tuples before processing them, we introduce *SUnion*, a simple data-serializing operator that takes multiple streams as



Bucket	Time t_0	Time t_1	Time t_2	Time t_3
tuple_stime ∈ [30,34]	Not stable	Not stable	Not stable	Not stable
tuple_stime ∈ [25,29]	Not stable	Not stable	Not stable	Stable
tuple_stime ∈ [20,24]	Not stable	Stable	Stable	Stable
tuple_stime ∈ [15,19]	Stable	Stable	Stable	Stable

Fig. 7. **Example showing bucket stability at different points in time. The SUnion has three input streams. The bucket size is 5 time units.**

inputs and orders all tuples deterministically into a single sequence. To ensure replica consistency, an SUnion is placed in front of each operator that takes multiple streams as input, and the latter operator processes tuples in the order defined by the SUnion. Using a separate operator enables the sorting logic to be contained within a single operator. More important, as it buffers tuples before sorting them, SUnion can also manage trade-offs between availability and consistency by deciding when tuples should be processed when failures occur. We discuss this second function of SUnion in later sections.

To manage availability-consistency trade-offs at a granularity coarser than individual tuples (thus greatly simplifying the implementation of SUnion), SUnion performs all its processing on buckets of input tuples. Each bucket covers a fixed and disjoint interval of tuple_stime values. SUnion uses tuple_stime values to place tuples in the appropriate bucket. SUnion then uses boundary tuples to determine when a bucket is *stable* (no more tuples will ever arrive for that bucket), at which time it is safe to order tuples in this bucket and output them. SUnion's sort function typically orders tuples by increasing tuple_stime values but other functions are possible. Assuming buckets of size 5 time-units, Figure 7 illustrates what buckets are stable at times t_0 through t_3 based on the boundary tuples received up to that point.

SUnion operators may appear at any location in a query diagram. All operators in the query diagram must thus set tuple_time values on their output tuples deterministically as these values will affect tuple order at downstream SUnions. Operators must also produce periodic boundary tuples with monotonically increasing tuple_time values.

SUnion is similar to the Input Manager in STREAM [Srivastava and Widom 2004], whose goal is to sort tuples by increasing timestamp order. In contrast, the goal of SUnion is to ensure that all replicas process tuples in the same order. SUnions thus need to appear in front of every operator with more than one input

and not just on the inputs to the system. More important, the Input Manager is not fault-tolerant. It assumes that delays are bounded and it uses that assumption to compute heartbeats if applications do not provide them. In contrast, as we discuss next, the SUnion operator handles failures by implementing the parameterizable availability/consistency trade-off and later participating in failure recovery.

4.2.2 *Impact of* tuple_stime *Values Selection.* A natural choice for tuple_stime is to use the local time at data sources. By synchronizing data source clocks, tuples will get processed approximately in the order in which they are produced. The Network Time Protocol (NTP) [The NTP Project] is standard today. It is implemented on most computers and essentially all servers, synchronizing clocks to within 10 milliseconds. Wall-clock time is not the only possible choice, though. In Borealis, any integer attribute can define the windows that delimit operator computations. When this is the case, operators also assume that input tuples are sorted on that attribute [Abadi et al. 2003]. Using the same attribute for tuple_stime as for windows helps enforce the ordering requirement.

SUnion delays tuples because it buffers and sorts them. This delay depends on three properties of boundary tuples. First, the interval between boundary tuples with increasing tuple_stime values and the bucket size determine the basic buffering delay. Second, the basic delay further increases with disorder. The increase is bounded above by the maximum delay between a tuple with a tuple_stime, t, and a boundary tuple with a tuple_stime $> t$. Third, a bucket is stable only when boundary tuples with sufficiently high tuple_stime values appear on all streams input to the same SUnion. The maximum differences in tuple_stime values across these streams bounds the added delay. Because the query diagram typically assumes that tuples are ordered on the attribute selected for tuple_stime, we can expect serialization delays to be small in practice. In particular, these delays should be significantly smaller than the maximum added processing delay, X.

In summary, the combination of SUnion operators and boundary tuples enables replicas of the same processing node to process tuples in the same order and remain mutually consistent. SUnions may increase processing latency because they buffer tuples before sorting and processing them, but this extra delay is small.

4.2.3 *Detecting Failures.* The heartbeat property of boundary tuples enables an SUnion to distinguish between the lack of data on a stream and a failure: when a failure occurs, an SUnion stops receiving boundary tuples on one or more input streams. SUnion may also start to receive tentative tuples.

Because we do not want to propagate tentative tuples through the query diagram as soon as they arrive but rather delay them based on the current availability requirement, and because SUnions serve to implement this delay, we place SUnion operators on each input stream to a processing node, even when the stream is the only input to an operator.

In addition to relying on boundary tuples to detect failures, the Consistency Manager monitors upstream neighbors and their replicas by periodically requesting a heartbeat response from *each replica* of each upstream neighbor. With this approach, if an upstream neighbor fails, the Consistency Manager knows the states of all replicas of that neighbor and can switch to using another replica. Heart-

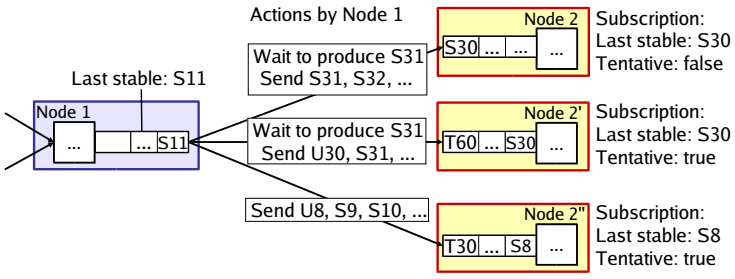


Fig. 8. **A node in STABLE state stabilizes the input streams of new downstream neighbors as per the information they provide in their subscriptions.** Subscription information is shown on the right of each downstream node (Node 2, Node 2′, and Node 2″). Actions taken by upstream Node 1 are shown on the arrows. U denotes an undo tuple and the identifier indicates the last tuple not to be undone. S denotes a stable tuple and T denotes a tentative tuple.

beat responses not only indicate if a replica is reachable, but also include the states (STABLE, UP_FAILURE, FAILURE, or STABILIZATION) of its output streams. These states can correspond to the overall consistency state of the node or they can be computed separately for each stream as we discuss in Section 8.2. The Consistency Manager computes these states.

4.3 UPSTREAM_FAILURE State

Each node must handle failures of its upstream neighbors in a manner that meets the application-required availability and ensures eventual consistency.

Because the Consistency Manager continuously monitors input streams, as soon as an upstream neighbor is no longer in the STABLE state (*i.e.*, it is either unreachable or experiencing a failure), the node can switch to another STABLE replica of that neighbor. By performing such a switch, the node can maintain both availability and consistency in spite of the failure. To enable the new upstream neighbor to continue sending data from the correct point in the stream, when a node switches replicas of an upstream neighbor, it indicates the *last stable tuple it received and whether it received tentative tuples after stable ones*. This information is provided by the Data Path to the Consistency Manager, which sends it to the new upstream neighbor in a subscribe message. The new upstream neighbor can then replay previously missing tuples or even correct previously tentative tuples. Data Paths at upstream neighbors must, of course, buffer their output tuples to perform such replays and corrections. We discuss buffer management in Section 8.1. Figure 8 illustrates the approach. In this Figure, Node 2 has only received stable tuples before losing its upstream neighbor. It thus indicates the identifier of the last stable tuple it received and Node 1 continues sending data from that point in time. Node 2' and Node 2" both already received tentative tuples after their last stable one. For this reason, Node 1 sends corrections for the previously tentative tuples. The undo tuple identifies the last tuple that should not be undone. It marks the beginning of the sequence of corrections.

If the Consistency Manager is unable to find a STABLE replica to replace an up-

Condition		Action
State(Curr(s), s)	R = Replicas(s) − Curr(s)	
STABLE	—	Do nothing
! STABLE	$\exists r \in R$, State(r, s) = STABLE	Unsubscribe from Curr(s) Curr$(s) \leftarrow r$ Subscribe to Curr(s)
UP_FAILURE	$\nexists r \in R$, State(r, s) = STABLE	Do nothing
∈ {FAILURE, STABILIZATION}	$\nexists r \in R$, State(r, s) = STABLE and $\exists r' \in R$, State(r', s) = UP_FAILURE	Unsubscribe from Curr(s) Curr$(s) \leftarrow r'$ Subscribe to Curr(s)
∈ {FAILURE, STABILIZATION}	$\nexists r \in R$, State(r, s) = STABLE and $\nexists r' \in R$, State(r', s) = UP_FAILURE	Do nothing

Table II. **Condition-action rules for switching replicas of an upstream neighbor in order to maintain availability.** When the condition on the left holds, the node takes the actions shown on the right. Replicas(s) is the set of all replicas producing stream s. Curr(s) is the current upstream neighbor for s. The state, State(Curr(s),s), of the stream s produced by Curr(s) and the states of the same stream produced by each replica in Replicas(s) define the conditions that can trigger an upstream neighbor switch. These switches in turn cause the state of Curr(s) to change. Input stream states also change as failures occur or heal.

stream neighbor, it should at least try to connect to a replica in the UP_FAILURE state because processing tuples from such a replica helps the node maintain availability. If the Consistency Manager cannot find a replica in either STABLE or UP_FAILURE states, the node cannot maintain the availability of the missing stream. Connecting to a replica in the STABILIZATION state allows the node to at least start correcting data on the failed stream. Table II presents the condition-action rules that nodes use to switch upstream replicas. Based on these rules, a node simply prefers upstream neighbors in STABLE state over those in UP_FAILURE, which it prefers over all others. The result of these switches is that any replica can forward data streams to any downstream replica or client and the outputs of some replicas may be unused. We refine the switching algorithm further after presenting the STABILIZATION state in Section 4.4.

In all cases, the node must process those input tuples that are available within the required time-bound. If a node proceeds with a missing input stream or processes tentative data, its state starts to diverge.

4.4 STABILIZATION State

A node determines that a failure healed when it is able to communicate with a stable upstream neighbor and receives corrections to previously-tentative tuples (or a replay of previously missing inputs). To ensure eventual consistency, the node must then reconcile its state and stabilize its outputs. This means that the node replaces previously tentative result tuples with stable ones, thus allowing downstream neighbors to reconcile their states in turn. To avoid correcting tentative tuples with other tentative ones, a node reconciles its state only after correcting at least some window of tuples on *all* its input streams. We present state reconciliation and output stabilization techniques in this section. We also present a technique that enables each node to maintain availability (meet the $\texttt{Delay}_{\texttt{new}} < X$ requirement) while reconciling its state.

4.4.1 *State Reconciliation.* Because no replica may have the correct state after a failure and because the state of a node depends on the exact sequence of tuples it

processed, we propose that a node reconcile its state by reverting it to a pre-failure state and reprocessing all input tuples since then. Different techniques are possible. We find that a technique based on checkpoint and redo performs well in most scenarios, outperforming an alternate technique based on undo and redo [Balazinska 2006]. With checkpoint and redo, a node checkpoints the state of its query diagram when it transitions into the UP_FAILURE state but before processing any tentative tuples. To perform a checkpoint, a node suspends all processing and iterates through operators and intermediate queues making a copy of their states. During the failure, SUnion operators placed on input streams buffer the input tuples they receive. To reconcile its state, the node then re-initializes operator and queue states from the checkpoint and reprocesses tuples buffered by input SUnions. To enable this approach, operators must be modified to include a method to take a snapshot of their state or reinitialize their state from a snapshot.

4.4.2 *Stabilizing Output Streams.* A node stabilizes each output stream by deleting a suffix of the stream (normally all tentative tuples) with a single undo tuple and forwarding corrections in the form of stable tuples. When it receives an undo tuple, an SUnion at a downstream node stabilizes the corresponding input stream by replacing, in its buffer, undone tuples with their stable counterparts. Once all input streams are corrected, SUnions trigger a state reconciliation.

To generate the undo tuple, we introduce a new operator, *SOutput*, that is placed on each output stream that crosses a node boundary. At runtime, SOutput acts as a pass-through filter that also remembers the last stable tuple it produced. During state reconciliation, SOutput drops duplicate stable tuples and produces the undo tuple when it finally sees the first new stable tuple or another tentative tuple.

Stabilization completes when one of two situations occurs. The node re-processes all previously tentative input tuples *and* catches up with normal execution (*i.e.*, it clears its queues) or another failure occurs and the node goes back into UP_FAILURE. In both cases, SUnion operators on input streams generate REC_DONE tuples, which propagate to the output streams. SOutput operators forward the REC_DONE tuples downstream and send a copy to the Consistency Manager. To avoid duplicate REC_DONE tuples, each SUnion operator placed in the middle of the diagram waits for a REC_DONE on all its inputs before forwarding a single such tuple downstream.

The above algorithms for state reconciliation and output stream stabilization enable a node to ensure eventual consistency: the node's state becomes once again consistent and downstream neighbors receive the complete and correct output streams. The problem, however, is that stabilization takes time and while reconciling its state and correcting its output, the node is not processing new input tuples. A long stabilization may cause the system to break the availability requirement. We discuss how to address this problem next.

4.4.3 *Processing New Tuples During Reconciliation.* Because the reconciliation itself may take longer than X time-units, a node cannot suspend new tuples while reconciling its state. It must produce both corrected stable tuples and new tentative tuples. We propose to achieve this by using two replicas of a query diagram: one replica remains in UP_FAILURE state and continues processing new input tuples while the other replica performs the reconciliation. A node could run both versions

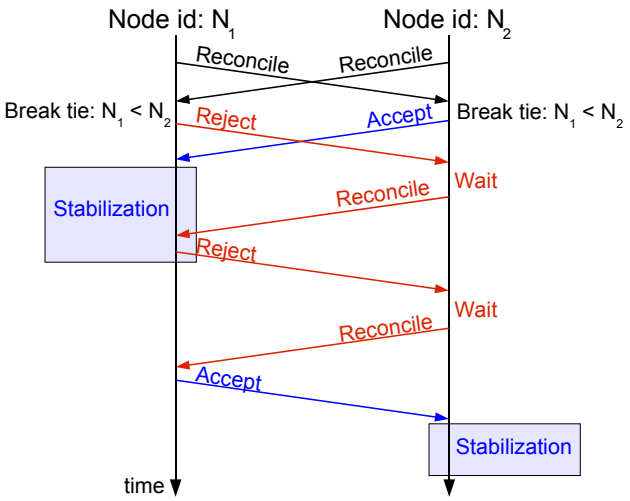


Fig. 9. **Inter-replica communication protocol to stagger replica stabilizations.**

locally but because we already use replication, we propose that replicas use each other as the two versions, when possible. By doing so, we never create new replicas in the system. Hence, to ensure availability, before reconciling its state, a node must find another replica and request that it postpone its own reconciliation. Only if the replica accepts, does the node enter the STABILIZATION state.

To enable a pair of replicas to decide which one should reconcile its state while the other one remains available, Consistency Managers run the following simple inter-replica communication protocol. Before entering STABILIZATION, the Consistency Manager sends a message to one of its randomly selected replicas. The message requests authorization to enter the STABILIZATION state. If the partner grants the authorization, it promises that it will not enter STABILIZATION itself. Upon receiving authorization to reconcile, the Consistency Manager triggers state reconciliation. However, if the replica rejects the authorization, the node cannot enter the STABILIZATION state. Instead, the Consistency Manager waits for a short time-period and tries to request authorization again, possibly communicating with a different replica. A Consistency Manager always accepts reconciliation requests from its replicas except if it is already in the STABILIZATION state or it needs to reconcile its own state and its identifier is lower than that of the requesting node. The latter condition is a simple tie breaker when multiple nodes need to reconcile their states at the same time. Figure 9 illustrates the communication taking place between two replicas that both need to reconcile their states.

It is up to each downstream node to detect when any one of its upstream neighbors goes into the STABILIZATION state. Upon entering the STABILIZATION state, the upstream neighbor stops producing recent tuples in order to produce corrections. The downstream node remains connected to that replica to correct its input stream in the background. At the same time, the node connects to another replica that is still in UP_FAILURE state (if possible) to continue processing new tentative

data and maintain availability. The downstream node stays connected to both upstream replicas until it receives a REC_DONE tuple on the corrected stream. At this point, the stable stream is up-to-date and the node can enter the STABILIZATION state in turn. SUnion considers that tentative tuples between an UNDO and a REC_DONE correspond to the old failure while tentative tuples that appear after the REC_DONE correspond to a new failure. The Data Path monitors input streams and ensures this property holds.

4.5 Failed Node Recovery

In the above sections, we described how the system recovers from a failure that causes one or more of a node's input streams to stop receiving data. In addition, of course, nodes may crash arbitrarily. To recover from a crash, a node starts from an empty state. Before it can consider itself in the STABLE state, the node must rebuild a consistent internal state and must catch-up with processing its input streams. Until it is STABLE, it must not reply to any requests, including heartbeats. Fortunately, rebuilding the state of a crashed node has been considered before in the literature and those solutions [Hwang et al. 2005; Shah et al. 2004] apply to our setting [Balazinska 2006].

5. DPC PROPERTIES

In this section, we evaluate the performance of DPC by conducting several experiments with our prototype implementation. Our first goal is to show that DPC provides eventual consistency even when multiple failures overlap in time. Our second goal is to show that, with at least two replicas of a processing node, DPC maintains the required availability at all times.

In the following section (Section 6), we study the design decisions that must be made when deploying DPC. We show that these decisions influence the trade-offs between availability and consistency.

All single-node experiments were performed on a 3 GHz Pentium IV with 2 GB of memory running Linux (Fedora Core 2). Multi-node experiments were performed by running each pair of node replicas on a different machine. All machines were 1.8 GHz Pentium IVs or faster with more than 1 GB of memory.

5.1 Eventual Consistency Guarantees

We first show examples of how DPC ensures eventual consistency in the face of simultaneous failures and failures during recovery. We show that in both scenarios client applications eventually receive the stable version of all result tuples and that no stable tuples are duplicated.

We run a single processing node (no replica, no upstream and downstream neighbors) and we control its inputs directly. Our goal is to examine the output produced by the node and show that it eventually produces the complete and correct output stream. The node runs the query diagram shown in Figure 10, which produces on the output stream tuples with sequentially increasing identifiers. The query diagram, composed solely of SUnion operators that deterministically merge their inputs and possibly delay them (see Section 4.2), is intentionally simple to help us investigate the properties of DPC. We first cause a failure on input stream 1 ("Failure 1"). We then cause a failure on input stream 3 ("Failure 2"). We plot the

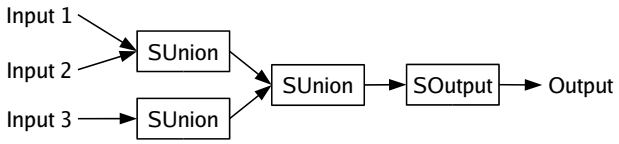
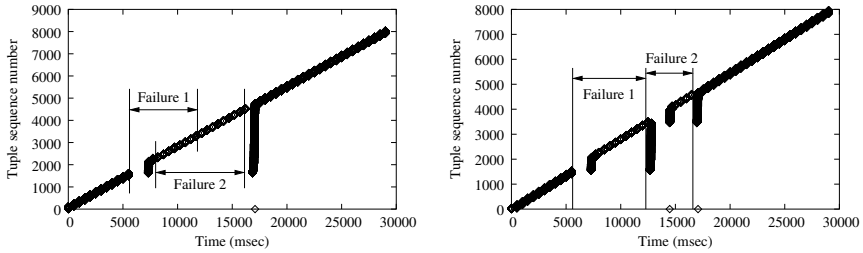


Fig. 10. **Query diagram used in simultaneous failures experiments.**



(a) Overlapping failures (b) Failure during recovery

Fig. 11. **Example outputs with simultaneous failures.**

sequence numbers received by the client application over time as the two failures occur and heal. Figure 11(a) shows the output when the two failures overlap in time. Figure 11(b) shows the output when Failure 2 occurs exactly at the moment when Failure 1 heals and the node starts reconciling its state.

In the case of simultaneous failures (Figure 11(a)), as Failure 1 occurs, the output first stops because the node suspends all processing. All tuples following the pause are tentative tuples. Nothing special happens when the second failure occurs because the output is already tentative. Nothing happens either when the first failure heals because the second failure is still occurring. It is only when all failures heal, that the node enters the STABILIZATION state and sends the stable version of previously tentative data. As the node finishes producing corrections and catches up with normal execution, it produces a REC_DONE tuple that we show on the figure as a tuple with identifier zero (tuple that appears on the x-axis). As the experiment shows, in our implementation, we chose the simpler approach of waiting for all failures to heal before reconciling the state. This approach works well when failures are infrequent. In the case of a large number of input streams and frequent failures, we could change the implementation to reconcile the state as soon as the first failure heals and the node can reprocess some window of stable input tuples on all input streams.

In the example with a failure during recovery (Figure 11(b)), as the first failure heals, the node enters STABILIZATION and starts producing corrections to previously tentative results. Before the node has time to catch-up with normal execution and produce a REC_DONE the second failure occurs and the node suspends processing once again. The node then produces a REC_DONE to indicate the end of the sequence of corrections before going back to processing tentative tuples once again. After the second failure heals, the node corrects *only those tentative tuples produced during the second failure*. The corrections are followed by a REC_DONE.

Hence all tentative tuples get corrected and no stable tuple is duplicated.

In the above experiments, the node temporarily lost one or two of its input streams, and there were no other replicas that it could reconnect to. When another replica exists for an upstream neighbor, the node will try to switch to that replica. When a node switches to a different replica of an upstream neighbor, there is a short gap in the data it receives. In our prototype implementation, we measured that it takes a node approximately 40 milliseconds to switch between upstream neighbors once the node detects a failure. Failure detection time depends on the frequency with which the downstream node sends keep-alive requests to its upstream neighbors. With a keep-alive period of 100 milliseconds, for example, it thus takes at most 140 milliseconds between the moment a failure (network failure or node failure) occurs and the moment a downstream node receives data from a different replica of its upstream neighbor. For many application domains, we expect this value to be much smaller than the minimum incremental processing latency that the application can tolerate. If an application cannot tolerate even such a short delay, the effect of switching upstream neighbors is the same as the effect of Failure 1 shown in Figure 11(a), but without the subsequent Failure 2: i.e., the downstream node first suspends; then produces tentative tuples; once reconnected to the new upstream neighbor, the downstream node goes back and corrects the tentative tuples it produced during the switch.

5.2 Availability Guarantees

In the above experiments, the maximum incremental latency, D, was set to 2 seconds.[3] As Figure 11 shows, the maximum gap between new tuples remains below that bound at any time. However, the node manages to maintain the required availability only thanks to a sufficiently short reconciliation time. For longer-duration failures, reconciliation could easily take longer than the maximum latency bound. In general, DPC relies on replication to enable a distributed SPE to maintain a low processing latency by ensuring at least one replica always processes the most recent input data within the required bound.

To demonstrate this property, we use the following experimental setup. We use two Borealis nodes with the same query diagram composed of three input streams, an SUnion that merges these streams into one, an SJoin with a 100-tuple state size, and an SOutput. Figure 12 illustrates our experimental setup (we use the same setup in the following section where we discuss availability and consistency trade-offs).

Our measure of availability is $\texttt{Delay}_{\texttt{new}}$, the maximum *added* processing latency for any new output tuple (see Section 2). However, because we have only one output stream in the experiment, we measure directly the maximum processing latency $\texttt{Proc}_{\texttt{new}} = \texttt{Delay}_{\texttt{new}} + \texttt{proc}(t)$. We create a failure by temporarily disconnecting one of the input streams without stopping the data source. After the failure heals, the data source replays all missing tuples while continuing to produce new tuples. Table III shows $\texttt{Proc}_{\texttt{new}}$ measured at the client for failures with different durations. Because the experiment is deterministic, each result is an average of only three

[3]In our implementation, because operators do not control when the scheduler invokes them, SUnion operators delay by $0.9D$ instead of D, as a precaution.

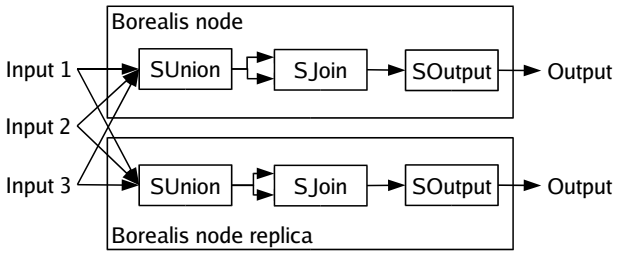


Fig. 12. **Experimental setup for experiments investigating the consistency and availability trade-offs for a single node.**

Failure duration (seconds)	2	4	6	8	10	12	14	16	30	45	60
$\texttt{Proc}_{\texttt{new}}$ (seconds)	2.2	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8

Table III. $\texttt{Proc}_{\texttt{new}}$ **for different failure durations and for a single-node deployment with one replica.** Independently of failure duration, $\texttt{Proc}_{\texttt{new}}$ remains constant and below the required threshold of 3 seconds.

experiments. All three values measured were within a few percent of the reported average.

As the table shows, independent of failure duration, the client always receives new data within the required 3-second bound.[4] Indeed, with this basic version of DPC, called Process & Process, each node processes input tuples as they arrive without trying to delay them to reduce inconsistency. More specifically, when the failure first occurs, both replicas suspend for the maximum incremental processing bound, then return to processing tentative tuples as they arrive. After the failure heals, DPC ensures that only one replica at a time reconciles its state while the remaining replica continues processing the most recent input data. Only once the first replica reconciles its state and catches up with current execution does the other replica reconcile its state in turn. The client application has thus access to the most recent data at all times.

6. AVAILABILITY AND CONSISTENCY TRADE-OFFS

In addition to ensuring eventual consistency while maintaining availability, another goal of DPC is to try to minimize inconsistency measured with $\texttt{N}_{\texttt{tentative}}$, the number of tentative tuples received by the client application.

In this section, we study the design decisions that achieve different desired trade-offs between availability and consistency. First, we study what SUnions can do with newly arriving tuples during UP_FAILURE and STABILIZATION. Indeed, SUnions can choose to either process tuples directly as they arrive (which ensures availability at the expense of consistency), suspend processing new tuples (which ensures consistency at the expense of availability), or they can try to delay new tuples in both states to maximize consistency while meeting the given availability bound. We perform this study first for a single replicated processing node (Section 6.1) and then

[4]As we discuss in the next section, we increased the maximum processing latency from 2 seconds to 3 seconds for this configuration to better emphasize availability-consistency trade-offs in the single-node case.

for a sequence of replicated processing nodes (Section 6.2). We find that processing or delaying tuples are the only valid alternatives for long-duration failures. We also find that delaying tuples continuously is much less helpful than expected as soon as a deployment includes more than one processing node in sequence. Given our findings from Sections 6.1 and 6.2, we study, in Section 6.3, how best to divide the maximum incremental processing latency specified by an application between the many SUnions in a query diagram. We show that assigning the entire delay, minus a small safety factor for queuing delays, to each SUnion yields the best overall trade-off between availability and consistency.

Another important design decision for DPC lies in the choice of boundary interval and SUnion bucket size. In general, these intervals must be significantly smaller than the maximum incremental processing latency. We study the impact of their exact values when we discuss the overhead of the approach in Section 7.

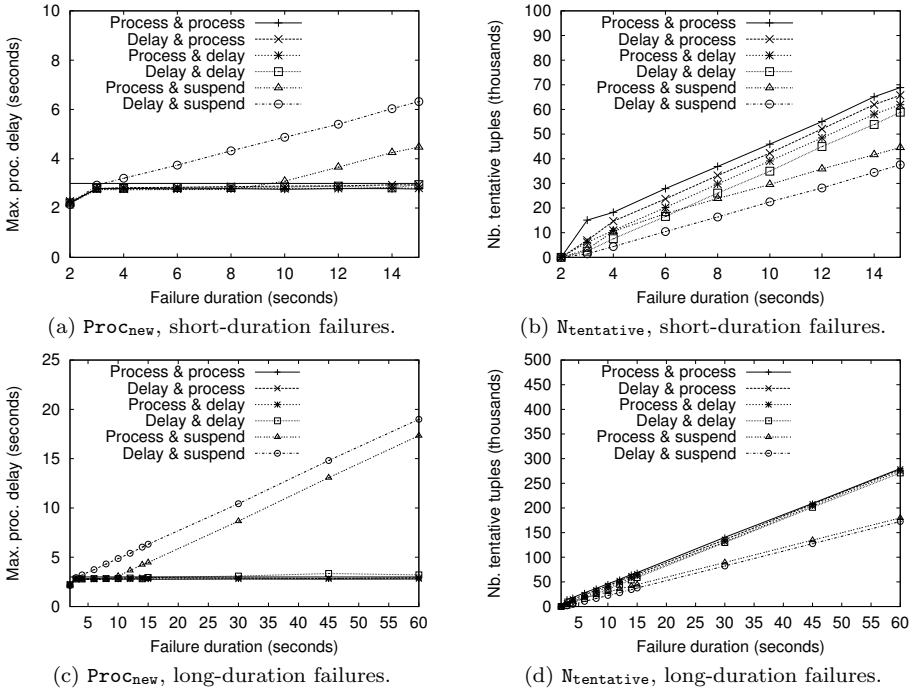
Finally, a system operator must also decide where to place replicas of processing nodes and how many replicas to use. DPC requires at least two replicas for each processing node (including the original node). Additional replicas improve availability at the expense of greater overall resource utilization. Replicas must, of course, be spread over the network to ensure their failures and disconnections exhibit as little correlation as possible and that replicas are placed close to clients. Properly placing replicas of servers on the Internet is a well-known problem (*e.g.*, [Radoslavov et al. 2001; Tang et al. 2007]) and we do not investigate it in this paper.

6.1 Single Node and Replica

In DPC, the application-defined maximum incremental processing latency, X, is divided among SUnions. Each SUnion is assigned some, possibly distinct, maximum delay bound, D. An SUnion can reduce inconsistency by delaying tuples more or less within this bound, D. We investigate such different variants of DPC in this section. For clarity of exposition, each processing node runs a query diagram with a single SUnion.

During UP_FAILURE, a node can suspend processing new input tuples until the maximum delay D expires. For failures that lasts longer than the threshold D, a node can either continuously delay tuples up-to D (we call this variant "Delay") or process them without delay ("Process"). During STABILIZATION, a node can either suspend new tuples ("Suspend"), or have a second version of the SPE continue processing them with or without delay ("Delay" or "Process"). Because delaying tuples during UP_FAILURE affects the results of suspending or delaying tuples during STABILIZATION, we examine all six possible combinations: (1) delaying or (2) processing new tuples during UP_FAILURE then (a) suspending, (b) delaying, or (c) processing new tuples during STABILIZATION. Our goal is to determine the failure durations when each combination of techniques produces the fewest tentative tuples without violating the low processing latency requirement. As we point out, some combinations are not viable as they break the low processing-latency requirement for sufficiently long-duration failures.

To better emphasize the differences between approaches, we increase the number of tentative tuples produced in the experiments by increasing the aggregate input rate to 4500 tuples/second and by increasing D to 3 seconds. The dynamics of failure and recovery remain the same, but the overall latency and number of tentative



(a) $\texttt{Proc}_{\texttt{new}}$, short-duration failures.

(b) $\texttt{N}_{\texttt{tentative}}$, short-duration failures.

(c) $\texttt{Proc}_{\texttt{new}}$, long-duration failures.

(d) $\texttt{N}_{\texttt{tentative}}$, long-duration failures.

Fig. 13. **Availability and consistency resulting from delaying, processing, or suspending new tuples during** UP_FAILURE **and** STABILIZATION**.** X-axis starts at 2 seconds. All techniques completely mask failures that last 2 seconds or less. Delay & Delay meets the availability requirement for all failure durations, while producing the fewest tentative results.

tuples increase when D and the input data rate increase.

Since there is a single processing node, the incremental processing latency per node, D, is equal to X, the total incremental latency tolerable by the application. Furthermore, in this experiment, $\texttt{Delay}_{\texttt{new}} < X$ when $\texttt{Proc}_{\texttt{new}} < 3$ seconds because the normal processing latency is below 300 milliseconds and, in our implementation, nodes always delay for $0.9D$ instead of D (as mentioned above),

Figure 13 shows $\texttt{Proc}_{\texttt{new}}$ and $\texttt{N}_{\texttt{tentative}}$ for each of the six variants that we consider and for different failure durations. We only show results for failures up to 1 minute. Longer failures continue the same trends. Because suspending is optimal for short-duration failures, all approaches suspend for time-period D, and produce no tentative tuples for failures below this threshold. Each result is the average of three experiments. Once again, all three values measured were within a few percent of the reported average. For Delay & Delay , the variation was a little higher. This variation is not intrinsic to the approach. It is due to the Borealis scheduler, which does not smoothly handle load spikes.

As indicated in the previous section, processing new tuples without delay during UP_FAILURE and STABILIZATION (Process & Process) ensures that the maximum delay remains below D independent of failure duration. Indeed, with this combination, at least one node always processes new tuples directly as they arrive (except

for an initial delay of less than D when the failure first occurs). This baseline combination, however, produces the most tentative tuples as it produces them for the duration of the whole failure and reconciliation. The SPE can reduce the number of tentative tuples without hurting $\texttt{Proc}_{\texttt{new}}$, by continuously delaying new tuples during STABILIZATION (Process & Delay), during UP_FAILURE (Delay & Process), or in both states (Delay & Delay). As expected, this last combination produces the fewest tentative tuples.

We now examine what happens if a node chooses to suspend rather than simply delay processing new tuples at some point during the failure and recovery process. First, as we mentioned above, suspending processing new tuples during failures is viable only for failures shorter than D. For longer-duration failures, however, it is possible to process tuples as they arrive during failures in order to have time to suspend processing new tuples during reconciliation (Process & Suspend). This approach is viable only when reconciliation is shorter than D. Otherwise, while the node is reconciling, $\texttt{Delay}_{\texttt{new}}$ exceeds D. In our experiment, this happens for a failure duration around 8 seconds. With Process & Suspend, we can thus save a number of $\texttt{N}_{\texttt{tentative}}$ tuples proportional to the reconciliation time times the tuple rate. The reconciliation time must always be equal to or less than D for the approach to be viable. In contrast, by delaying (Delay & Delay) instead of suspending (Process & Suspend), the savings is equal to exactly D times the tuple rate. Delay & Delay is thus always equivalent or better than Process & Suspend. The latter combination is thus uninteresting.

Finally, delaying tuples in UP_FAILURE then suspending them during STABILIZATION (Delay & Suspend) is not viable because delaying during failures brings the system on the verge of breaking the availability requirement. Suspending during reconciliation then adds a delay proportional to the reconciliation time.

In summary, to meet the availability requirement for long-duration failures, nodes *must* process new tuples not only during UP_FAILURE but also during STABILIZATION. Nodes can produce fewer tentative tuples, however, by always running on the verge of breaking that requirement (Delay & Delay).

6.2 Distributed Setting

We now examine the performance of the above techniques in a distributed setting. Because it is never advantageous to suspend processing tuples during reconciliation and because suspending breaks the availability requirement for long-duration failures, we only compare two techniques: continuously delaying new tentative tuples (Delay & Delay) and processing tentative tuples almost as they arrive (Process & Process). We expected that delaying tuples as much as possible would result in a longer processing latency but would lead to fewer tentative tuples on the output stream. We show that in contrary to our expectations, delaying tentative tuples does not improve consistency, except for short-duration failures.

Figure 14 shows the experimental setup: a chain of up to four processing nodes, each replicated. As in the previous section, the first SUnion merges the inputs from three streams. Subsequent SUnions process a single input stream. Each pair of processing nodes runs on a different physical machine. The data sources, first pair of processing nodes, client proxy, and client all run on the same machine.

To cause a failure, we temporarily prevent one of the input streams from pro-

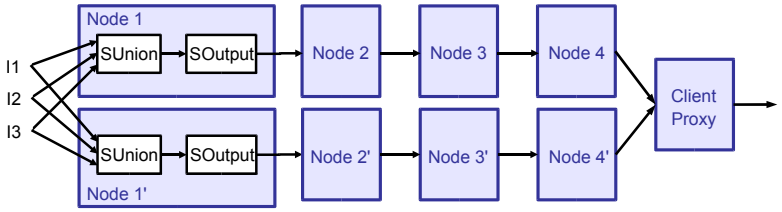


Fig. 14. **Setup for experiments with a distributed SPE.**

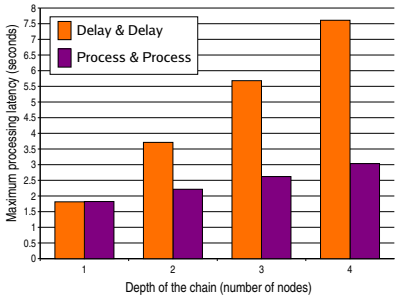
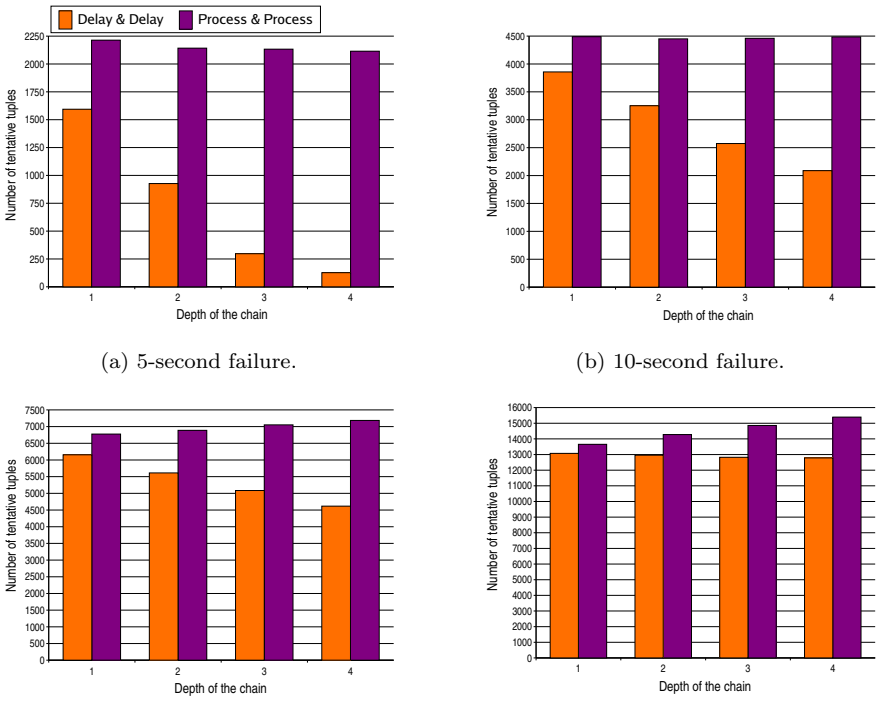


Fig. 15. **$\texttt{Proc}_{\texttt{new}}$ for a sequence of processing nodes.** Each node runs a single SUnion with $D = 2$ seconds. Results are independent of failure durations. Each result is the average of 10 experiments (standard deviations, σ, are within 3% of means). Both techniques meet the required availability of 2 seconds per-node. Process & Process provides a significantly better availability.

ducing boundary tuples. We choose this technique rather than disconnecting the stream to ensure that the output rate at the end of the chain is the same with and without the failure. Keeping the output rate the same makes it easier to understand the dynamics of the approach in this setting. The aggregate input rate is 500 tuples/second. We cause failures of different durations between 5 seconds and 60 seconds.

Figure 15 shows the measured availability of the output stream during UP_FAILURE and STABILIZATION as we increase the depth of the chain. We show results only for a 30-second failure because the end-to-end processing latency is independent of failure duration, except for very short-duration failures. In this experiment, we assign a maximum delay, $D = 2$ seconds, to each SUnion. The end-to-end processing latency requirement thus increases linearly with the depth of the chain. It is $2n$ seconds for a chain of n nodes. Both Delay & Delay and Process & Process provide the required availability, but the availability is significantly better with Process & Process.

For Delay & Delay, each node in the chain delays its input tuples by D before processing them. The processing latency thus increases by a fixed amount for every consecutive processing node. For Process & Process, we would expect the *maximum* processing latency to be the same. As a failure occurs and propagates through the



(a) 5-second failure.

(b) 10-second failure.

(c) 15-second failure.

(d) 30-second failure.

Fig. 16. $N_{tentative}$ **during *short-duration* failures and reconciliations.** Each node runs one SUnion with $D = 2$ seconds. Each result is the average of 10 experiments (standard deviations, σ, are within 4% of means except for the 5-second failure with Delay & Delay, where σ is about 16% of the mean for deep chains). In the presence of short failures, delaying improves consistency only by a fixed amount approximately proportional to the total delay through the chain.

chain, each node that sees the failure first delays tuples by D, before processing *subsequent* tuples almost as they arrive. The first bucket without a boundary tuple should thus be delayed by each node in sequence. Instead, for Process & Process, the end-to-end processing latency is closer to the delay imposed by a single processing node. Indeed, as the failure occurs, *all nodes suspend processing at the same time* because when the first node suspends processing it also suspends producing boundary tuples; all nodes stop receiving boundary tuples at the same time. Thus, after the initial 2-second delay, tuples stream through the rest of the chain with only a small extra delay per node. This observation is important because it affects the assignment of delays to SUnions, as we see later in this section. In summary, Process & Process achieves significantly better availability (latency), although both techniques meet our requirement.[5]

[5]The processing latency increases by a small amount per-node even with Process & Process because, in the current implementation, SUnions do not produce tentative boundaries. Without boundaries, an SUnion does not know how soon a bucket of tentative tuples can be processed. We currently require SUnions to wait for a minimum of 300 milliseconds before processing a tentative bucket. Using tentative boundaries would enable even faster processing of tentative data and

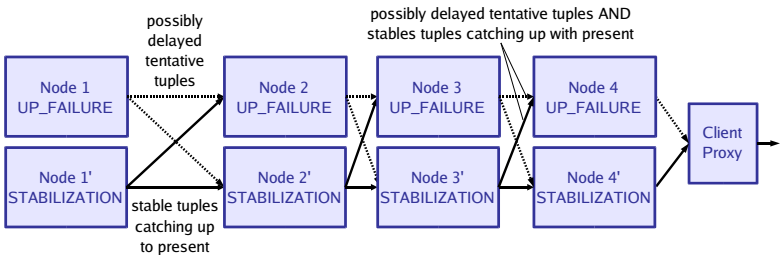


Fig. 17. **Dynamics of state reconciliation through a chain of nodes.** When a failure heals, at least one replica from each set enters the STABILIZATION state. Every node thus receives simultaneously potentially delayed tentative tuples and stable tuples, which slowly catch-up with current execution.

We now compare the number of tentative tuples produced on the output stream to examine how much delaying helps improve consistency (*i.e.*, reduces $\mathtt{N_{tentative}}$).

Figure 16 shows $\mathtt{N_{tentative}}$ measured on the output stream for each technique and failures up to 30 seconds in duration. In these examples, failures are relatively short, reconciliation is relatively fast, and delaying tentative tuples reduces inconsistency. For a given failure duration, the consistency gain is approximately proportional to the total delay through the chain of nodes: *i.e.*, the gain increases with the depth of the chain. However, this delay is independent of failure duration, which means that the relative gains actually *decrease* with failure duration (*i.e.*, as reconciliation gets longer). At 30 seconds, the gains start to become insignificant.

To discuss the results in detail, we must examine the dynamics of state reconciliation through a chain of nodes. In the implementation, a node enters the STABILIZATION state as soon as one previously tentative bucket becomes stable. As a failure heals, the first node starts reconciling its state and starts producing corrections to previously tentative tuples. Almost immediately, the downstream neighbors receive enough corrections for at least one bucket to become stable, and one of the downstream neighbors enters the STABILIZATION state. Hence, in a chain of nodes, as soon as a failure heals, *at least one replica of each node starts to reconcile its state*. These replicas form a chain of nodes in the STABILIZATION state. The other replicas form a parallel chain of nodes that remain in the UP_FAILURE state, as illustrated in Figure 17. This approach has two important effects.

First, the total reconciliation time increases only slightly with the depth of the chain because all nodes reconcile roughly at the same time. For Process & Process, the number of tentative tuples is proportional to the failure duration plus the stabilization time. For 30-second and 15-second failures, we clearly see the number of tentative tuples increase slightly with the depth of the chain.[6]

latency would remain approximately constant with the depth of the chain (increasing only as much as the actual processing latency).

[6]A few tentative tuples are typically dropped when a node switches upstream neighbors. For 5-second failures, the number of tentative tuples decreases with the depth of the chain even for Process & Process because these drops are not yet offset by increasing reconciliation times.

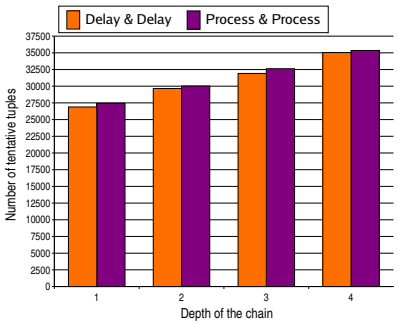


Fig. 18. $\mathtt{N_{tentative}}$ **during a *long-duration* failure and reconciliation.** Each node runs a single SUnion with $D = 2$ seconds. Each result is the average of 10 experiments (standard deviations, σ, are within 3% of means). For long-duration failures, delaying does not improve consistency.

Second, because a whole sequence of nodes reconciles at the same time, the last replica in the chain that remains in UP_FAILURE state receives both delayed tentative tuples from its upstream neighbor in UP_FAILURE state and *most recent corrected stable tuples* from the upstream neighbor in the STABILIZATION state. The last node in the chain processes these most recent stable tuples as tentative because it is still in the UP_FAILURE state. For short-duration failures, reconciliation is sufficiently fast that the last node in UP_FAILURE state does not have time to get to these most recent tuples before the end of STABILIZATION at its replica. The last node in the chain only processes the delayed tentative tuples. Therefore, for Delay & Delay and short-duration failures, the number of tentative tuples decreases with the depth of the chain. It decreases proportionally to the total delay imposed on tentative tuples.

The result is different for long-duration failures, though. Figure 18 shows $\mathtt{N_{tentative}}$ on the output stream when the failure lasts 60 seconds. The figure shows that the benefits of delaying almost disappear. Delay & Delay can still produce a little fewer tentative tuples than Process & Process but the gain is negligible and independent of the depth of the chain. The gain is equal to only the delay, D, imposed by the last node in the chain. Therefore, for long-duration failures, delaying sacrifices availability without benefits to consistency.

In summary, in a distributed SPE, the best strategy for processing tentative tuples during failure and reconciliation is to first suspend all processing hoping that failures are short. If failures persist past the delay, D, the new best strategy is to continuously delay new input tuples as much as possible. As the failure persists and an increasingly large number of tuples will have to be re-processed during state reconciliation, delaying is no longer beneficial, and nodes might as well improve availability by processing tuples without any delay. This technique could easily be automated, but as we show later an even better strategy exists.

6.3 Assigning Delays to SUnions

In the previous sections, we assumed that each SUnion was assigned a fixed delay, D. We now examine how to divide an end-to-end maximum added processing latency, X, among the many SUnions in a query diagram.

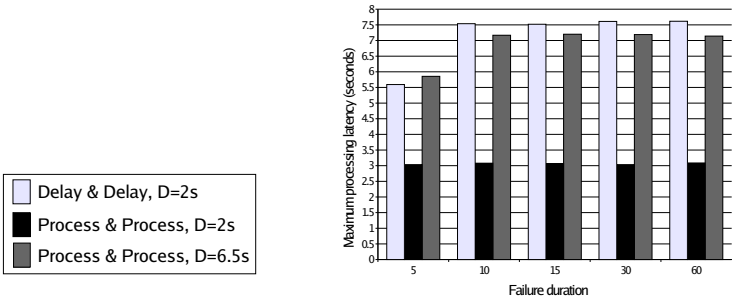
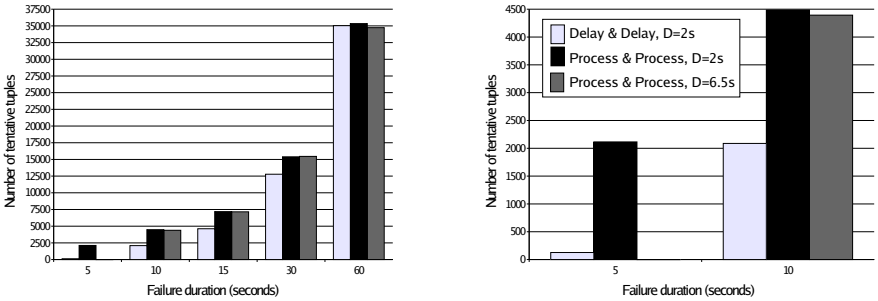


Fig. 19. $\texttt{Proc}_{\texttt{new}}$ **for a sequence of four processing nodes and different failure durations.** Each result is the average of 10 experiments (standard deviations, σ, are within 3% of means). It is possible to assign the full incremental delay to each SUnion in a chain and still meet the availability requirement.

We first study a chain configuration and examine two different delay assignment techniques: uniformly dividing the available delay among the SUnions in the chain or assigning each SUnion the *total incremental delay*. For a total incremental delay of 8 seconds, and a chain of four processing nodes, the first technique assigns a delay of 2 seconds to each node. This is the technique we have been using until now. In contrast, with the second technique, we assign the complete 8-second delay to each SUnion. In the experiments, we actually use 6.5 seconds instead of 8 seconds because queues start to form when the initial delay is long. The 6.5 second value was selected to approximately take this queuing delay into account.

Figure 19 shows the maximum processing latency for a chain of 4 nodes and the two different delay assignment techniques. Figure 20(a) shows $\texttt{N}_{\texttt{tentative}}$ for the same configurations. The left and middle bars repeat the results from the previous section: each SUnion has $D = 2$ seconds and either delays tentative tuples or processes them without delay. The graphs on the right show the results when each SUnion has $D = 6.5$ seconds and processes tuples without delay.

Interestingly, assigning the total delay to each SUnion meets the required availability independently of the depth of the chain. Indeed, when a failure occurs, *all SUnions downstream from the failure suspend at the same time*. After the initial delay, however, nodes must process tuples as they arrive to meet the availability requirement (alternatively, they could revert to delaying by only 2 seconds). At first glance, though, assigning the total delay to each SUnion appears to have the worse availability of Delay & Delay and the worse number of tentative tuples of Process & Process. Figure 20(b) shows a close-up on the results for only the 5-second and 10-second failures. For the 5-second failure, when $D = 6.5$ seconds, the system does not produce even one tentative tuple. Assigning the maximum incremental processing latency to each SUnion thus enables a system to cope with the longest possible failures without introducing inconsistency and still meeting the required availability. Additionally, the high maximum processing latency when assigning the whole delay to each SUnion affects only the tuples that enter the system *as the failure first occurs*. After the initial delay, nodes process subsequent tuples without



(a) Various failure durations. (b) Short-duration failures only.

Fig. 20. $\texttt{N}_{\texttt{tentative}}$ **for different delay assignments to SUnions.** Process & Process with $D = 6.5$ seconds is the only technique that can mask the 5-second failure while performing as well as the other approaches for longer failures. Each result is the average of 10 experiments (standard deviations, σ, are within 11% of means).

any delay. The availability thus goes back to the low processing latency of Process & Process.

In a more complex graph configuration, the same result holds. When an SUnion first detects a failure, all downstream SUnions suspend processing tuples at the same time. The initial delay can thus be equal to X. After the initial delay, however, SUnions can either process tuples without further delay, or they can continuously delay new tuples for a shorter incremental time, D. We have shown that additive delays are not useful for long-duration failures. In a graph configuration, they are also difficult to optimize. Figure 21 illustrates the problem. Every time two streams meet at an operator, their respective accumulated delays depend on the location of upstream failures and the number of SUnions they traversed. There is therefore no single optimal incremental delay that SUnions can impose *individually*, even when delays are assigned separately to each input stream at each SUnion. As shown on the figure, it is possible to produce an assignment guaranteeing that the SPE meets a required availability, but some failure configurations can cause some tentative tuples to be systematically dropped by an SUnion. Pre-defined additive delays are thus undesirable. To circumvent this problem, the SPE could encode accumulated delays inside tuples and SUnions could dynamically adjust the incremental delays they impose based on the total accumulated delays so far. This type of assignment would lead to a new set of possible delay assignment optimizations, but we do not examine such delay assignments in this paper.

In this section, we investigated trade-offs between availability and consistency for a single SPE and for simple distributed deployments. We showed that in order to minimize inconsistency while meeting a desired availability level, when first detecting a failure, each SUnion should suspend processing tuples for as long as the total incremental latency specified by the application. After this initial delay, SUnions should process tuples as they arrive because independent incremental delays do not improve consistency after long-duration failures and, for complex query diagrams, may cause a greater number of tuples to be dropped because of mismatches in accumulated delays on different streams.

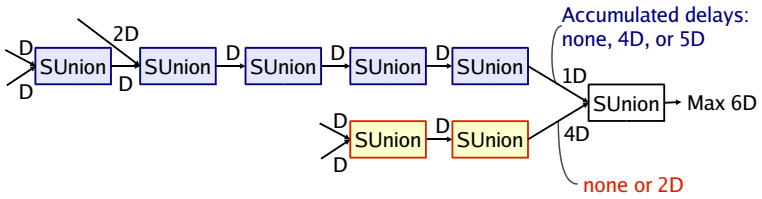


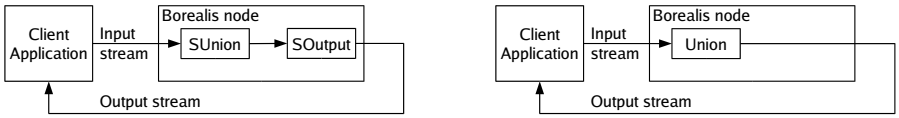
Fig. 21. **Assigning incremental delays to SUnions in a query diagram.** The assignment is difficult because accumulated delays depend on the diagram structure and on the failure location.

7. DPC OVERHEAD

The main overhead of DPC lies in buffering tuples during failures in order to replay them during state reconciliation. This overhead is a memory overhead; it does not affect runtime performance, and we discuss it in the next section. There are, however, additional sources of overhead that affect runtime performance. We discuss these overheads in this section.

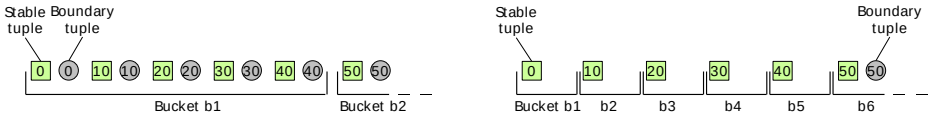
Tuple serialization is the main cause of runtime overhead. If the sort function requires an SUnion to wait until a bucket is stable before processing tuples in that bucket, the processing delay of each SUnion increases linearly with the boundary interval and the bucket size. To evaluate the overhead of these delays, we use the experimental setup shown in Figure 22. The data source generates one input tuple approximately every 10 milliseconds. We run the experiment for 5 minutes, producing a total of approximately 25,000 tuples in each run. For all these tuples, we measure the minimum, maximum, and average per-tuple processing latency. We run this experiment for varying bucket sizes and boundary intervals. We vary each parameter independently to show how each parameter affects the processing latency. Figure 23 shows examples of streams for two specific parameter configurations. In the actual experiments, the data source paused for 10 milliseconds between tuples. As a result, tuples were produced a little slower than shown in the figure. Tables IV and V show the results from these experiments. In both tables, the column with zero delay shows the result when we replace the SUnion and SOutput combination with a standard Union operator and we remove all boundary tuples from the stream. The latency values shown in Table IV are slightly lower than those in Table V because, for boundary intervals of 10 milliseconds, we simply send a boundary tuple directly with each data tuple. As expected, the maximum and average processing latency increase proportionally to both the boundary interval and the bucket size. Both parameters must thus be set to small values to avoid unnecessarily delaying tuples.

In addition to processing latency, tuple serialization also introduces memory overhead since SUnions buffer tuples before sorting them. This memory overhead increases proportionally to the number of SUnion operators, their bucket sizes, and the rate of tuples that arrive on each input stream to an SUnion. Because we typically chose the tuple_stime to correspond to the expected order of tuples on input streams, we could significantly reduce both memory and latency overheads by allowing SUnions to output tuples as soon as they receive input tuples with higher tuple_stime values on all their input streams. This optimization, however, requires that all operators process and produce tuples in increasing tuple_stime order.



(a) Fault-tolerant setup (b) Setup without fault-tolerance

Fig. 22. **Experimental setup used in fault-tolerance overhead experiments.**



(a) Bucket size of 50 milliseconds and boundary interval of 10 milliseconds

(b) Bucket size of 10 milliseconds and boundary interval of 50 milliseconds

Fig. 23. **Illustration of data streams for two sample experimental configurations.**

Other overheads imposed by DPC do not affect performance in a measurable way. These overhead include operators checking tuple types and processing boundary tuples. The former is negligible while the latter requires, in the worst case, a scan of all tuples in the operator's state, and, in the best case, requires simply that the operator propagates the boundary tuple. The operators that we modified need not scan their state to process boundary tuples, ensuring that the overhead of these boundary tuples is negligible. As a third source of overhead, each SOutput must also save the last stable tuple that it sees in every burst of tuples that it processes.

Finally, DPC relies on replication. It increases resource utilization proportionally to the number of replicas. These replicas, however, can actually improve runtime performance by forming a content distribution network, where clients and nodes connect to nearby upstream neighbors rather than a single, possibly remote, location.

8. DISCUSSION

In this section, we discuss additional details of DPC: we discuss buffer management and we outline a technique for handling failures at a finer granularity.

8.1 Buffer Management

For DPC to work, the data path must buffer output tuples and SUnions must buffer input tuples as follows:

Output Buffers: A node must buffer the output tuples it produces until all replicas of all downstream neighbors receive these tuples. Indeed, at any point in time, any replica of a downstream neighbor can connect to any replica of an upstream neighbor and request all input tuples it has not yet received. Output buffers are thus needed during failures, but they are also needed in the absence of failures because nodes do not process and transmit data at the same time. We assume that nodes have spare processing and bandwidth capacity (see Section 2.2). Therefore, nodes can keep up with input rates without falling behind. Nevertheless, some nodes may run somewhat ahead of others nodes. The nodes that are ahead

Bucket size (ms)	0	10	50	100	150	200	300	500
Minimum latency	0	12	12	12	13	13	13	14
Maximum latency	5	26	64	113	165	213	313	514
Average latency	0	13.3	31.1	56.6	81.5	106.5	156.6	258.0
Standard deviation of latency	0	1.9	14.5	28.7	43.1	57.5	86.2	144.3

Table IV. **Latency overhead of serialization.** Varying bucket size for a fixed boundary interval of 10 milliseconds. The column with 0 milliseconds bucket size shows the results when using a standard Union instead of SUnion and SOutput and when not producing boundary tuples at all.

Boundary interval (ms)	0	10	50	100	150	200	300	500
Minimum latency	0	12	14	15	17	19	20	25
Maximum latency	5	26	70	121	170	219	317	520
Average latency	0	13.3	37.3	62.1	87.0	111.6	166.2	269.4
Standard deviation of latency	0	1.9	16.6	30.4	43.7	56.9	87.3	141.9

Table V. **Latency overhead of serialization.** Varying boundary interval for a fixed bucket size of 10 milliseconds. The column with 0 milliseconds boundary interval shows the results when using a standard Union instead of SUnion and SOutput and when not producing boundary tuples at all.

must buffer their output tuples until the other nodes produce the same output tuples and forward them downstream.

Input Buffers: In the absence of failures, no data is buffered on input streams. When failures occur, nodes need to buffer the stable input tuples they receive in order to reprocess them later during STABILIZATION: *i.e.*, SUnions placed on input streams need to buffer tuples they receive after the pre-failure checkpoint because reconciliation involves restarting from the last checkpoint and reprocessing all input tuples received since then.

Overall, in the absence of failures, DPC requires only a small bounded amount of buffer space: nothing is buffered on the inputs and output buffers can be truncated in response to periodic acknowledgments from downstream neighbors. However, if downstream nodes crash or become disconnected they may no longer send acknowledgments, forcing *all buffers to grow with the duration of the failure*.

Limiting buffer sizes may prevent nodes from reconciling their states and correcting all previously tentative results after a sufficiently long failure. Tuples in buffers could be written to disk so even without additional mechanisms, DPC could tolerate relatively long-duration failures. We want, however, to limit the size of all buffers to keep buffers in memory speeding-up their replay and bounding the time it takes nodes to recover from failures. We propose two different techniques depending on the type of operators in the query diagram.

Deterministic Operators: The state of a deterministic operator can, in the worst case, depend on all tuples that the operator ever processed. With such operators, any tuple loss during a failure may prevent nodes from becoming consistent again. Such a situation is called system delusion [Gray et al. 1996]: replicas are inconsistent and there is no obvious way to repair the system. To avoid system delusion, we propose to maintain availability only as long as there remains space in buffers. Once a node's buffers fill up, the node blocks. Blocking creates back pressure all the way up to the data sources, which start dropping tuples without pushing them into the system. This technique maintains availability only as long as

buffer sizes permit but it ensures eventual consistency. It avoids system delusion.

Convergent-Capable Operators: Convergent-capable operators [Hwang et al. 2005; Balazinska 2006] have the nice property that any input tuple affects their state only for a bounded amount of time. Convergent capable operators include but are not limited to all stateless operators (*e.g.*, Filter and Map) and value-based sliding window Joins and Aggregates. When a query diagram consists only of these types of operators, we can compute, for any location in the query diagram, a maximum buffer size, S, that guarantees enough tuples are being buffered to *rebuild the latest consistent state and correct the most recent tentative tuples*. With this approach, the system can support failures of arbitrarily long duration with bounded-size buffers. Users can indicate the window of most-recent tuples that need to be corrected after a failure heals (*e.g.*, the last hour or the last day) and the system can compute the necessary buffer sizes to ensure this property, while maintaining availability at all times.

Convergent-capable query diagrams are thus more suitable for applications that need to maintain availability during failures of arbitrarily long duration, yet also need to reach a consistent state after failure heals. For other deterministic operators, buffer sizes determine the duration of failures that the system can support while maintaining availability.

8.2 Reducing Failure Granularity

Until now, we assumed that a node advertises a single overall consistency state to its downstream neighbors and that the whole local query diagram restarts from a checkpoint during `STABILIZATION`. Frequently, however, a failure affects only a subset of the operators running at a node. Ideally, we would like to limit the impact of failures and recovery to these operators only. We propose to do so by computing and advertising the state of each output stream separately and by recovering the state of operators individually.

To advertise the individual states of its output stream, a node must compute these states. One approach is for each SOutput operator to determine the state of its output stream by observing if tuples are stable or tentative. A better approach is for the Consistency Manager to compute that state from the set of failures on input streams and the structure of the query diagram. This state can be computed even before tentative tuples propagate to the output. In both cases, the state of output streams unaffected by a failure remains advertised as STABLE, and the corresponding downstream neighbors do not need to switch to another replica.

Second, to limit the impact of failure recovery, a node can reconcile its state by restarting only those operators that processed tentative data during the failure. In previous work [Balazinska et al. 2005; Balazinska 2006], we noted that restarting an operator from a checkpoint is faster and simpler than undoing and redoing its state. We thus propose to use this approach but avoid node-wide checkpoints and recovery. Instead, we propose that each operator checkpoints its state individually right before processing its first tentative tuple. During STABILIZATION, SUnions on previously failed inputs can push an undo tuple into the query diagram before replaying the stable tuples. When an operator receives an undo tuple, it recovers its state from its checkpoint and reprocesses all stable tuples that follow the undo. With this approach, only operators that see a failure checkpoint their state and the

only SUnions that need to buffer tuples are those with both tentative and stable inputs, and they only need to buffer tuples on the stable inputs (the only tuples they are in charge of replaying). Additionally, only those operators that previously experienced a failure participate in the state reconciliation thus effectively limiting the impact of failures.

9. RELATED WORK

Until now, work on high availability in stream processing systems has focused on fail-stop failures of processing nodes [Hwang et al. 2005; Shah et al. 2004]. These techniques either do not address network failures [Hwang et al. 2005] or strictly favor consistency by requiring at least one fully connected copy of the query network to exist to continue processing [Shah et al. 2004]. Some techniques use punctuation [Tucker and Maier 2003], heartbeats [Srivastava and Widom 2004], or statically defined slack [Abadi et al. 2003] to tolerate bounded disorder and delays. These approaches, however, block or drop tuples when disorder or delay exceed expected bounds. Another approach, developed for publish-subscribe systems tolerates failures by restricting all processing to "incremental monotonic transforms" [Strom 2004]. DPC has no such restriction.

Traditional query processing also addresses trade-offs between result speed and consistency, materializing query outputs one row or even one cell at the time [Naughton et al. 2001; Raman and Hellerstein 2002]. In contrast to these schemes, our approach supports possibly infinite data streams and ensures that once failures heal all replicas produce the same final output streams in the same order.

Fault-tolerance through replication is widely studied and it is well known that it is not possible to provide both consistency and availability in the presence of network partitions [Brewer 2001]. Eager replication favors consistency by having a majority of replicas perform every update as part of a single transaction [Garcia-Molina and Barbara 1985; Gifford 1979] but it forces minority partitions to block. With lazy replication all replicas process possibly conflicting updates even when disconnected and must later reconcile their state. They typically do so by applying system- or user-defined reconciliation rules [Kawell et al. 1988; Urbano 2003], such as preserving only the most recent version of a record [Gray et al. 1996]. It is unclear how one could define such rules for an SPE and reach a consistent state. Other replication approaches use tentative transactions during partitions and reprocess transactions possibly in a different order during reconciliation [Gray et al. 1996; Terry et al. 1995]. With these approaches, all replicas eventually have the same state and that state corresponds to a single-node serializable execution. Our approach applies the ideas of tentative data to stream processing.

Some schemes offer users fine-grained control over the trade-off between precision (or consistency) of query results and performance (*i.e.*, resource utilization) [Olston 2003; Olston et al. 2003]. In contrast, we explore consistency/availability trade-offs in the face of failures and ensure eventual consistency.

Workflow management systems (WFMS) [Alonso et al. 1995; Alonso and Mohan 1997; Hsu 1995] share similarities with stream processing engines. Existing WFMSs, however, typically commit the results of each execution step (or messages these

steps exchange) in a central highly-available storage server [Kamath et al. 1996] or in persistent queues [Alonso et al. 1995; Alonso et al. 1997]. Because the data transferred between execution steps can be large, some WFMSs use a separate data manager [Alonso et al. 1997]. The data manager is a distributed and replicated DBMS and has thus the same properties as the eager or lazy replication schemes discussed above. Some WFMS also support disconnection by locking activities prior to disconnection [Alonso et al. 1995].

Approaches that reconcile state after a failure using combinations of checkpoints, undo, and redo are well known [Elnozahy et al. 2002; Gray et al. 1996; Gray and Reuters 1993; Lomet and Tuttle 2003; Terry et al. 1995]. We adapt and use these techniques in the context of fault-tolerance and state reconciliation in an SPE.

Real-time systems must guarantee that tasks complete by specific deadlines [Kao and Garcia-Molina 1995; Lam et al. 2000]. This requirement can be seen as analogous to our maximum incremental processing latency requirement. The goal of real-time systems, however, is to provide fault-tolerance with minimal overhead and impact on the processing predictability and performance. Real-time systems typically rely on custom code [Melliar-Smith and Moser 2004] and slack in their schedules [Mossé; et al. 2003] to achieve this goal. Additionally, real-time systems are often embedded systems and do not address the problem of network partitions. In contrast, our goal is to process best-effort data during network partitions and maximally exploit the application tolerance for latency increase in order to reduce inconsistency during these types of failures.

10. CONCLUSION

In this paper, we presented DPC, a replication-based approach to fault-tolerant stream processing that handles node failures, network failures, and network partitions. DPC uses a new data model that distinguishes between stable tuples and tentative tuples, which result from processing partial inputs and may later be corrected. DPC favors availability over consistency, but guarantees eventual consistency. Additionally, while ensuring that each node processes new tuples within a pre-defined delay, D, our approach reduces the number of tentative tuples when possible. To ensure consistency at runtime, we introduce a data-serializing operator called SUnion. To regain consistency after failures heal, nodes reconcile their states using checkpoint and redo.

We implemented DPC in Borealis and showed several experimental results. For short-duration failures, SPE nodes can avoid inconsistency by blocking and looking for a stable upstream neighbor. For long-duration failures, nodes need to process new inputs both during failure and stabilization to ensure the required availability.

Many stream processing applications prefer approximate results to long delays but eventually need to see the correct output streams. It is important that failure-handling schemes meet this requirement. We view this work as an important step in this direction.

11. ACKNOWLEDGMENTS

We thank Barbara Liskov, Mehul Shah, and Jeong-Hyon Hwang for helpful discussions. We also thank the anonymous reviewers for their help and comments on early

drafts of this paper (and on drafts of the preceding conference paper [Balazinska et al. 2005]). This material is based upon work supported by the National Science Foundation under Grant No. 0205445. M. Balazinska was partially supported by a Microsoft Research Fellowship.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. 2005. The design of the Borealis stream processing engine. In *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. 2003. Aurora: A new model and architecture for data stream management. *VLDB Journal 12,* 2 (Sept.).

Aleri. `http://www.aleri.com/index.html`.

Alonso, G., Günthör, R., Kamath, M., Agrawal, D., El Abbadi, A., and Mohan, C. 1995. Exotica/FMDC: Handling disconnected clients in a workflow management system. In *Proc. of the 3rd International Conference on Cooperative Information Systems*.

Alonso, G. and Mohan, C. 1997. WFMS: The next generation of distributed processing tools. In *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds. Kluwer.

Alonso, G., Mohan, C., Günthör, R., Agrawal, D., El Abbadi, A., and Kamath, M. 1995. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. of IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*.

Alonso, G., Reinwald, B., and Mohan, C. 1997. Distributed data management in workflow environments. In *Proc. of the 7th ACM RIDE*.

Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbets, R., and Zdonik, S. 2004. Retrospective on Aurora. *VLDB Journal 13,* 4 (Dec.).

Balazinska, M. 2006. Fault-tolerance and load management in a distributed stream processing system. Ph.D. thesis, Massachusetts Institute of Technology.

Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. 2005. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*.

Bernstein, P. A., Hsu, M., and Mann, B. 1990. Implementing recoverable requests using queues. In *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*.

Brewer, E. A. 2001. Lessons from giant-scale services. *IEEE Internet Computing 5,* 4, 46–55.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., and Zdonik, S. 2003. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*.

Coral8. `http://coral8.com/`.

Cranor, C., Johnson, T., Shkapenyuk, V., and Spatscheck, O. 2003. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*.

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey 34,* 3, 375–408.

Fekete, A., Gupta, D., Luchangco, V., Lynch, N., and Shvartsman, A. 1996. Eventually-serializable data services. In *Proc. of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC 1996)*.

Garcia-Molina, H. and Barbara, D. 1985. How to assign votes in a distributed system. *Journal of the ACM 32,* 4 (Oct.), 841 – 860.

Gifford, D. K. 1979. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles (SOSP).*

Gilbert, S. and Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News 33,* 2.

Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data.*

Gray, J. and Reuters, A. 1993. *Transaction processing: concepts and techniques.* Morgan Kaufmann.

Hsu, M. 1995. Special issue on workflow systems. *IEEE Data Engineering Bulletin 18,* 1 (Mar.).

Hwang, J.-H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., and Zdonik, S. 2005. High-availability algorithms for distributed stream processing. In *Proc. of the 21st International Conference on Data Engineering (ICDE).*

Kamath, M., Alonso, G., Guenthor, R., and Mohan, C. 1996. Providing high availability in very large workflow management systems. In *Proc. of the 5th International Conference on Extending Database Technology.*

Kao, B. and Garcia-Molina, H. 1995. An overview of real-time database systems. In *Advances in real-time systems.* Prentice-Hall.

Kawell, L., Beckhardt, S., Halvorsen, T., Ozzie, R., and Greif, I. 1988. Replicated document management in a group communication system. In *Proc. of the 1988 ACM conference on computer-supported cooperative work (CSCW).*

Lam, K.-W., Son, S. H., Hung, S.-L., and Wang, Z. 2000. Scheduling transactions with stringent real-time constraints. *Information Systems 25,* 6 (Sept.), 431–452.

Lomet, D. and Tuttle, M. 2003. A theory of redo recovery. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data.*

Melliar-Smith, P. M. and Moser, L. E. 2004. Progress in real-time fault tolerance. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04).*

Mossé;, D., Melhem, R., and Ghosh, S. 2003. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Transactions on Software Engineering 29,* 8.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR).*

Naughton et al. 2001. The Niagara Internet query system. *IEEE Data Engineering Bulletin 24,* 2 (June).

Olston, C. 2003. Approximate replication. Ph.D. thesis, Stanford University.

Olston, C., Jiang, J., and Widom, J. 2003. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data.*

Radoslavov, P., Govindan, R., and Estrin, D. 2001. Topology-informed Internet replica placement. In *Proc. of the Sixth International Workshop on Web Caching and Content Distribution (WCW'01).*

Raman, V. and Hellerstein, J. M. 2002. Partial results for online query processing. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data.*

Saito, Y. and Shapiro, M. 2005. Optimistic replication. *ACM Computing Surveys 37,* 1, 42–81.

Shah, M., Hellerstein, J., and Brewer, E. 2004. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data.*

Srivastava, U. and Widom, J. 2004. Flexible time management in data stream systems. In *Proc. of the 23rd ACM Symposium on Principles of Database Systems (PODS).*

StreamBase. http://www.streambase.com/.

Strom, R. E. 2004. Fault-tolerance in the SMILE stateful publish-subscribe system. In *Proc of the 3rd International Workshop on Distributed Event-Based Systems (DEBS '04)*.

Tang, X., Chi, H., and Chanson, S. T. 2007. Optimal replica placement under TTL-based consistency. *IEEE Transactions on Parallel and Distributed Systems 18,* 3, 351–363.

Terry, D. B., Theimer, M., Petersen, K., Demers, A. J., Spreitzer, M., and Hauser, C. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*.

The NTP Project. NTP: The Network Time Protocol. http://www.ntp.org/.

Tucker, P. A. and Maier, D. 2003. Dealing with disorder. In *Proc of the Workshop on Management and Processing of Data Streams (MPDS)*.

Urbano, R. 2003. *Oracle Streams Replication Administrator's Guide, 10g Release 1 (10.1)*. Oracle Corporation.