A Latency and Fault-Tolerance Optimizer for Online Parallel Query Plans

Prasang Upadhyaya
University of Washington
prasang@cs.uw.edu

YongChul Kwon
University of Washington
yongchul@cs.uw.edu

Magdalena Balazinska
University of Washington
magda@cs.uw.edu

ABSTRACT

We address the problem of making online, parallel query plans fault-tolerant: *i.e.*, provide *intra-query fault-tolerance without blocking*. We develop an approach that not only achieves this goal but does so through the use of *different fault-tolerance techniques at different operators within a query plan*. Enabling each operator to use a different fault-tolerance strategy leads to a space of fault-tolerance plans amenable to cost-based optimization. We develop FTOpt, a cost-based fault-tolerance optimizer that automatically selects the best strategy for *each* operator in a query plan in a manner that minimizes the expected processing time with failures for the entire query. We implement our approach in a prototype parallel query-processing engine. Our experiments demonstrate that (1) there is no single best fault-tolerance strategy for all query plans, (2) often hybrid strategies that mix-and-match recovery techniques outperform any uniform strategy, and (3) our optimizer correctly identifies winning fault-tolerance configurations.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Fault tolerance, modeling techniques*; H.2.4 [**Database Management**]: Systems—*Parallel databases,Query processing*

General Terms

Performance

1. INTRODUCTION

The ability to analyze large-scale datasets has become a critical requirement for modern business and science. To carry out their analyses, users are increasingly turning toward parallel database management systems (DBMSs) [14, 41, 45] and other parallel data processing engines [10, 15, 20] deployed in shared-nothing clusters of commodity servers.

In many systems, users can express their data processing needs using SQL or other specialized languages (*e.g.*, Pig Latin [30], DryadLINQ [48]). The resulting queries or scripts are then translated into a directed acyclic graph (DAG) of operators (*e.g.*, relational operators, maps, reduces, or other [20]) that execute in the cluster.

An important challenge faced by these systems is fault-tolerance. When running a parallel query at large scale, some form of failure is likely to occur during execution [9]. Existing systems take two radically different strategies to handle failures: parallel DBMSs restart queries if failures occur during their execution. The limitation of this approach is that a single failure can cause the system to reprocess a query in its entirety. While this is not a problem for queries running across a small number of servers and for a short period of time, it becomes undesirable for long queries using large numbers of servers. In contrast, MapReduce [10] and similar systems [15] materialize the output of each operator and restart individual operators when failures occur. This approach limits the amount of work repeated in the face of failures, but comes at the cost of materializing all intermediate data, which adds significant overhead even in the absence of failures. Furthermore, because MapReduce materializes data in a blocking fashion, this approach prevents users from seeing results incrementally. Partial results are a desirable feature during interactive data analysis now commonly performed with these systems [43].

In this paper, we study the problem of providing users both the ability to see early results as motivated by online query processing [17, 18] and achieve a low expected total runtime. We thus seek to *enable intra-query fault-tolerance without blocking* and we want to do so in a manner that *minimizes the expected total runtime in the presence of failures*. Other objective functions could also be useful (*e.g.*, minimize runtime without failures subject to a constraint on recovery time.) We choose to minimize the sum of time under normal processing and time in failure recovery. This function combines high-performance at runtime with fast failure recovery into a single objective. We want to minimize this function while preserving pipelining.

Recent work [43] has also looked at the problem of combining pipelining and fault-tolerance: they developed techniques for increased data pipelining in MapReduce. This system *both* pipelines and materializes data between operators. We observe, however, that data materialization is only one of several strategies for achieving fault-tolerance in a pipelined query plan. Other strategies are possible including restarting a query or operator but skipping over previously processed data [21, 24] or checkpointing operator states and restarting from these checkpoints [11, 21]. Additionally, the most appropriate fault-tolerance method may depend on the available resources, failure rates, and query plan properties. For example, an expensive join operator may need to checkpoint its state while an inexpensive filter may simply skip over previously processed data after a failure.

Given these observations, we develop (1) a framework that enables *mixing-and-matching of fault-tolerance techniques in a single, pipelined query plan* and (2) *FTOpt, a cost-based fault-tolerance optimizer* for this framework. Our framework enables intra-query fault-tolerance *without blocking*, thus preserving pipelining. Given a query plan and information about the cluster and expected failure rates, FTOpt automatically selects the fault-tolerance strategy for each operator in a query plan such that the overall query runtime with failures is minimized. We call the resulting configuration a *fault-tolerance plan*. In our fault-tolerance plans, each operator can individually recover after failure and it can recover using a different strategy than other operators in the same plan. In summary, we make the following contributions:

1. *Extensible, heterogeneous fault-tolerance framework.* We propose a framework that enables the mixing and matching of different fault-tolerance techniques in a single distributed, parallel, and pipelined query plan. Our framework is extensible in that it is agnostic of the specific operators and fault-tolerance strategies used. We also describe how three well-known strategies can be integrated into our framework (Section 4).
2. *Fault-tolerance optimizer.* We develop a cost-based fault-tolerance optimizer. Given a query plan and a failure model, the optimizer selects the fault-tolerance strategy *for each operator* that minimizes the total time to complete the query given an expected number of failures (Section 5).
3. *Operator models for pipelined plans.* We model the processing and recovery times for a small set of representative operators. Our models capture operator performance *within a pipelined query plan* rather than in isolation. They are sufficiently accurate for the fault-tolerance optimizer to select good plans yet sufficiently simple for global optimization using a Geometric Program Solver [3]. We also develop an approach that simplifies the modeling of other operators within our framework thus simplifying extensibility (Section 5.3).

We implemented our approach in a prototype parallel query processing engine. The implementation includes our new fault-tolerance framework, specific per-operator fault-tolerance strategies for a small set of representative operators (select, join, and aggregate[1]), and a MATLAB module for the FTOpt optimizer. Our experiments demonstrate that different fault-tolerance strategies, often hybrid ones, lead to the best performance in different settings: for the configurations tested, total runtimes with as little as *one* failure differed by up to 70% depending on the fault-tolerance method selected. These results show that fault-tolerance can significantly affect performance. Additionally, our optimizer is able to correctly identify the winning fault-tolerance strategy for a given query plan. Overall, FTOpt

[1]As in online aggregation [18], aggregates can occur at top of plans. Our prototype uses a standard aggregate operator but it could be replaced with an online one.

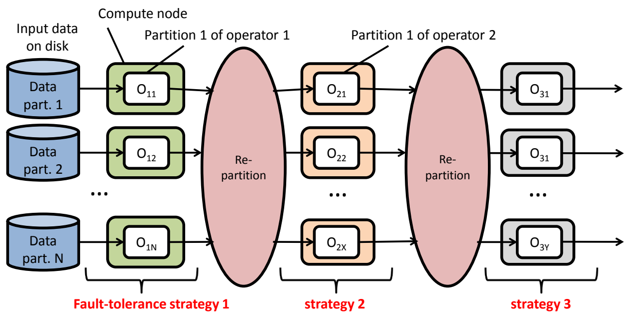

Figure 1: Parallel query plan comprising three operators (O_1, O_2, O_3) and one input from disk. Each operator is partitioned across a possibly different number of nodes. Data can be re-partitioned between operators. Fault-tolerance strategies are selected at the granularity of operators.

is thus an important component of parallel data processing, enabling performance gains similar in magnitude to several other recently proposed MapReduce optimizations [22, 26].

2. MODEL AND ASSUMPTIONS

Query Model and Resource Allocation. In parallel data processing systems, queries take the form of directed acyclic graphs (DAGs) of operators that are distributed across servers in a cluster as illustrated in Figure 1. Servers are also referred to as *nodes*. Each operator can be partitioned and these partitions then execute in parallel on the same or on different nodes. Multiple operators can also share the same nodes. In this paper, we focus on *non-blocking query plans*, which take the form of trees (rather than DAGs), where operators are scheduled and executed at the same time, and where data is pipelined from one operator to the next, *producing results incrementally*. We assume that aggregation operators, if any, appear only at the top of a plan. Since input data comes from disk, it can be read and consumed at a steady pace (there are no unexpected bursts as in a streaming system for example). If a query plan is too large for all operators to run simultaneously, our approach will optimize fault-tolerance for pipelined subsets of the plan, materializing results at the end.

Fault-tolerance choices and resource allocation are intertwined: an operator can perform more complex fault-tolerance if it is allocated a greater fraction of the compute resources. In addition to fault-tolerance strategies, our optimizer computes the appropriate allocation of resources to operators. Due to space constraints, however, in this paper, we assume that each operator is partitioned across a given number of compute nodes and is allocated its own core(s) and disk on that node. For the resource allocation details, we refer the reader to Appendix A.5.

Failure Model. In a shared-nothing cluster, different types of failures occur. Our approach handles a variety of failures from process failures to network failures. To simplify the presentation, we first focus on *process failures*: *i.e.*, we assume that each operator partition runs in its own process and that these processes crash and are then restarted (with an empty state) independently of one another. We come back to more complex failures in Section 5.5.

To make fault-tolerance choices, our optimizer must know the likelihood for different types of failures. If n_i is the total number of processes allocated to operator i, we assume that the expected number of failures during query execution for that operator is given by: $z_i = \frac{n_i}{n} Z$ where $n = \sum_{j \in \mathcal{O}} n_j$, $\mathcal{O}$ is the set of all operators in the plan, and Z is the expected number of process failures for the query. Z can be estimated from the observed failure rates for previous queries and administrators typically know this number [9]. We assume Z to be independent of the chosen fault-tolerance plan. Z depends on the query runtime, whose order of magnitude can be estimated by FTOpt as the total runtime without fault-tolerance and without failures (we show that results are robust to small errors in Z's value in Section 6.6).

Operator Determinism. We assume that individual operator partitions are deterministic, *i.e.*, an operator partition produces an identical output when it processes the same input tuples in the same order. This is a common assumption [20, 47, 37, 2, 36, 23] and most relational operators are deterministic. In a distributed system, however, the order in which input tuples reach an operator partition may not be deterministic. Our approach handles this scenario.

3. RELATED WORK

Fault-Tolerance in Relational DBMSs. Commercial relational DBMSs provide fault-tolerance through replication [6, 34, 40]. Similarly, parallel DBMSs [14, 41, 45] use replication to handle various types of failures. Neither, however, provides intra-query fault-tolerance [32].

Main-memory DBMSs [25, 35, 28] use a variety of checkpointing strategies to preserve the in-memory state of their databases. In contrast, our approach preserves and recovers the state of ongoing computations.

Fault-Tolerance in MapReduce-type systems. The MapReduce framework [10] provides intra-query fault-tolerance by materializing results between operators and re-processing these results upon operator failures. This approach, however, imposes a high runtime overhead and prevents users from seeing any output until the job completes. In Dryad [20], data between operators can either be pipelined or materialized. In contrast, we strive to achieve both pipelining and fault-tolerance at the same time. We also study how to decide when to materialize or checkpoint data. Recent work [47] applies MapReduce-style fault-tolerance to distributed databases by breaking long-running queries into small ones that execute and can be restarted independently. This approach, however, supports only a specific type of queries over a star schema. In contrast, we explore techniques that are more generally applicable. Recent work also introduced the ability to partly pipeline data in Hadoop [43], a MapReduce-type platform. This work is complementary to ours as it retains the use of materialization throughout the query plan for fault-tolerance purposes.

Other Fault-Tolerance Strategies. In the distributed systems and stream processing literatures, several additional fault-tolerance strategies have been proposed [11, 21, 37]. All these strategies involve replication. One set of techniques is based on the *state-machine* approach. Here, the same computation is performed in parallel by two processing nodes [2, 36, 37]. We do not consider such techniques in this paper because of their overhead: to tolerate even a single failure, they require twice the resources. The second set of techniques uses *rollback recovery* methods [11, 21], where the system takes periodic snapshots of its state that it copies onto stable storage (*i.e.*, into memory of other nodes or onto disk). We show how to integrate the latter techniques into our fault-tolerance optimization framework (Section 4.2).

Recently, Simitsis et. al. [38] studied the problem of selecting fault-tolerance strategies and recovery points for ETL flows. Similar to us they consider using different fault-tolerance strategies within a single flow. In contrast to our work, they do not propose a general heterogeneous fault-tolerance framework, do not have individually recoverable operators, and do not optimize for overall latency nor show how fault-tolerance choices affect processing latencies.

Additional Related Work. Hwang *et al.* [19] studied self-configuring high-availability methods. Their approach is orthogonal to our work as it is based on a uniform checkpointing strategy and optimizes the time when checkpoints are taken and the backup nodes where they are saved.

Techniques for query suspend and resume [4, 5] use rollback recovery but are otherwise orthogonal to our work.

Phoenix/App [27] explores the problem of heterogeneous fault-tolerance in the context of web enterprise applications. This approach identifies three types of software components: Persistent, Transactional, and External depending on the fault-tolerance strategy that each uses (message logging with checkpointing, transactions, or nothing respectively). Phoenix/App then defines different "interaction contracts" for each combination of component types. Each contract implements a different protocol with different guarantees. Thus in Phoenix/App, the protocol depends on the fault-tolerance capabilities of the communicating components. In contrast, our approach enables the mixing-and-matching of fault-tolerance strategies without changes to the protocol.

4. FRAMEWORK FOR HETEROGENEOUS FAULT-TOLERANCE

We present a framework for mixing and matching fault-tolerance techniques. Our framework relies on concepts from the literature including logging, acknowledging, and replaying tuples as previously done in uniform fault-tolerance settings [21, 37] and "contract-based" methods for query suspend-resume [4]. Our contribution lies in *articulating how these strategies can be used to enable fault-tolerance heterogeneity.* We also discuss how three fault-tolerance techniques from the literature can be used within our framework.

4.1 Protocol

To enable heterogeneous fault-tolerance between consecutive operators in a query plan, we isolate these operators by fixing the semantics of their interactions through a set of four rules. These rules enable each operator to be *individually restartable* without requiring any blocking materialization as in MapReduce and also without requiring that all operators use the same fault-tolerance strategy.

In our framework, as in any parallel data processing system, operators receive input tuples from their upstream neighbors; they process these tuples and send results downstream. For example, in Figure 1, each partition of operator O_2 receives data from each O_1 partition and sends data to all O_3 partitions. If an operator partition such as O_{21} fails, a new instance of the operator partition is started with an empty state. To recover the failed state, in our framework, the new instance can read any state persistently captured by the operator's fault-tolerance strategy. It can also ask upstream operators to resend (a subset) of their data. To enable such replays, tuples must have unique identifiers, which may or may not be visible to applications, and operators must remember the output they produced. For this, we define the following two rules:

RULE 4.1. *Each relation must have a key.*

RULE 4.2. Producer replay guarantee. *Upon request, an operator, must regenerate and resend* in order *and* without duplicates *any subset of unacknowledged output tuples.*

Acknowledgments mentioned in this rule help reduce the potential overhead of storing old output tuples by bounding how much history must be retained [21, 37]. In our framework, acknowledgments are optional and are sent from downstream operators to upstream ones. For example, once all operator partitions O_{21} through O_{2X} that have received an input tuple t from operator partition O_{11} acknowledge this tuple, the tuple need no longer be retained by O_{11}. Upon sending an acknowledgment, an operator promises never to ask for the corresponding tuple again. Formally,

RULE 4.3. Consumer progress guarantee. *If an operator acknowledges a tuple r_x, it guarantees that, even in case of failure, it will never ask for r_x again.*

Most parallel data processing systems use in-order communication (*e.g.*, TCP) between operators. In that case, an operator can send a single message with the identifier of a tuple r_x to acknowledge all tuples up to and including r_x.

When a failure occurs and an operator restarts with an empty state, most fault-tolerance techniques will cause the operator to produce duplicate tuples during recovery. To ensure that an operator can eliminate duplicates before sending them downstream, we add a last requirement:

RULE 4.4. Consumer Durability Guarantee. *Upon request, an operator O_d must produce the identifier of the most recent input tuple that it has received from an upstream neighbor O_u.*

Together, these four rules enable a parallel system to ensure that it produces the same output tuples in the same order with and without failure (the tuples may still be delayed due to failure recovery.) They also enable operators to be individually restartable and the query plan to be both pipelined and fault-tolerant, since data can be transmitted at anytime between operators. Finally, the framework is agnostic of the fault-tolerance method used as long as the method works within the pre-defined types of interactions.

From the above four rules, only the "Producer replay guarantee" rule potentially adds a visible overhead to the system since it requires that a producer be able to re-generate (part of) its output.[2] A no-cost solution to satisfy this rule is for an operator to restart itself upon receiving a replay request. With this strategy, an operator failure can cause a cascading rollback effect, where all preceding operators in the plan get also restarted. This approach is equivalent to restarting a subset of the query plan after a failure occurs and is no worse than what parallel DBMSs do today. Alternatively, an operator could write its output to disk. Finally, some operators, such as joins, can also easily re-generate their output from their state without the need to log their output. Each of these solutions leads to different expected query runtimes with and without failures. Our optimizer is precisely designed to select the correct strategy for each operator (from a pre-defined set of strategies) in a way that minimizes the total runtime with failures for a given query plan as we discuss further below.

[2]Our framework also requires unique identifiers for tuples. In our implementation, we create unique identifiers consisting of 3 integers [44]; for the 512 byte tuples used in our experiments the space overhead is less than 2.5%.

4.2 Concrete Framework Instance

We now discuss how three fault-tolerance strategies from the literature can be integrated into our framework.

Even though the operators in our framework are deterministic (see Section 2), in a distributed setting tuples may arrive in different interleaved order on different inputs. We develop a low-overhead method –based on lightweight logging of information about input tuple processing order– to ensure determinism in this case, but we omit it due to space constraints and refer the reader to Appendix A.4.

Strategy NONE. Within our framework, an operator can choose to do nothing to make itself fault-tolerant. We call this strategy NONE. To ensure that it can recover from a failure, such an operator can simply avoid sending any acknowledgments upstream. Upon a failure, that operator can then request that its upstream neighbors replay their entire output. This strategy is analogous to the *upstream backup* approach developed for stream processing engines [21].

As in upstream backup, operators such as select or project that do not maintain state between consecutive tuples (*i.e.*, "stateless operators") can send acknowledgments in some cases: *e.g.*, if an input tuple r makes it through a selection to generate the output q and is acknowledged by all operators downstream, then r can be acknowledged. Unlike upstream backup, which uses different types of acknowledgments [21], our approach uses only one type of acknowledgments facilitating heterogeneous fault-tolerance. This approach of skipping over input data during recovery has also been used for resumptions of interrupted warehouse loads [24].

To handle a request for output tuples, a stateless operator can fail and restart itself to reproduce the requested data. For stateful operators (*i.e.*, operators such as joins that maintain state between consecutive tuples), a more efficient strategy is to maintain an output queue and replay the requested data [21]. Such a queue, however, can still impose a significant memory overhead and an I/O overhead if the queue is written to disk. We observe, however, that stateful relational operators *need not keep such output queue* but, instead, can re-generate the data from their state. We implement this strategy and use it in our evaluation.

Strategy MATERIALIZE. An alternate rollback recovery approach consists in logging intermediate results between operators as in MapReduce [10]. While CHCKPT speeds-up recovery for the checkpointed operator itself, MATERIALIZE potentially speeds-up recovery for downstream operators: to satisfy a replay request, an operator can simply re-read the materialized data. Since materialized output tuples need never be generated again, an operator can use the same acknowledgement and recovery policy as in NONE.

Strategy CHCKPT. This strategy is a type of *rollback recovery* strategy where operators' state is periodically saved to stable storage. Because our framework recovers operators individually, it requires what is called *uncoordinated checkpointing with logging* [11]. One approach that can directly be applied is *passive standby* [21], where operators take periodic checkpoints of their state, independently of other operators.

Our framework requires that an operator save sufficient information to guarantee the consumer progress, consumer durability, and producer replay guarantees. For this, the operator must log its state (*e.g.*, partial aggregates, join hash tables) and, when applicable, its output queue. The operator can acknowledge checkpointed input tuples. Upon failures, the operator restarts from the last checkpoint. As an optimization, operators can checkpoint only delta-changes of their state [11]. Other optimizations are also possible [11, 19, 23] and can be used with our framework.

Unlike NONE and MATERIALIZE, with CHCKPT blocking operators also benefit from fault-tolerance provisioning as they can checkpoint their state periodically and restart from the latest checkpoint after a failure.[3]

In summary, while our framework imposes constraints on operator interactions, all three of these common fault-tolerance strategies can easily be incorporated into it.

5. FTOpt

FTOpt is an optimizer for our heterogeneous fault-tolerance framework. FTOpt runs as a post-processing step: it takes as input (a) a query plan selected by the query optimizer and annotates it with the fault-tolerance strategies to use. The optimizer also takes as input (b) information about the cluster resources and cluster failure model, and (c) models for the operators in the plan under different fault-tolerance strategies. FTOpt produces as output a fault-tolerance plan that minimizes an objective function (*i.e.*, the expected runtime with failures) given a set of constraints (that model the plan).

FTOpt's fault-tolerance plans have three parts: (1) a fault-tolerance strategy for each operator, (2) checkpoint frequencies for all operators that should checkpoint their states, and (3) an allocation of resources to operators. As indicated in Section 2, however, we do not discuss resource allocation in this paper due to space constraints. We assume a given resource allocation to operators.

For a given query plan, the optimizer's search space thus consists of all combinations of fault-tolerance strategies. In this paper, we use a brute-force technique to enumerate through that search space and leave more efficient enumeration algorithms for future work. For each such combination, FTOpt estimates the expected total runtime with failures, the optimal checkpoint frequencies and, optionally, an allocation of resources to operators [44]. It then chooses the plan with the minimum total runtime with failures.

5.1 Geometric Model

As it enumerates through the search-space, given a potential fault-tolerance plan, in order to select optimal checkpoint frequencies and estimate the total runtime with failures for the plan, FTOpt uses a geometric programming (GP) framework. GP allows expressions that model resource scaling (for resource allocation) and non-linear operator behavior, but still finds a global minima for the model [3].

In a geometric optimization problem, the goal is to minimize a function $f_0(\vec{x})$, where $\vec{x}$ is the optimization variable vector. The optimization is subject to constraints on other functions $f_i(\vec{x})$ and $g_i(\vec{x})$. All of $\vec{x}$, $g(\vec{x})$, and $f(\vec{x})$ are constrained to take the following specific forms:

- $\vec{x} = (x_1, \dots, x_n)$ such that $\forall i \;\; x_i > 0, x_i \in \mathbb{R}$.
- $g(\vec{x})$ must be a monomial of the form $cx_1^{a_1}x_2^{a_2}\dots x_n^{a_n}$ with $c > 0$ and $a_i \in \mathbb{R}$.
- $f(\vec{x})$ must be a posynomial defined as a sum of one or more monomials. Specifically, with $c_k > 0$ and $a_{ik} \in \mathbb{R}$: $f(\vec{x}) = \sum_{k=1}^{k=K} c_k x_1^{a_{1k}} x_2^{a_{2k}} \dots x_n^{a_{nk}}$.

The optimization is then expressed as follows:

$$\begin{array}{rll} \text{minimize} & f_0(\vec{x}) & \\ \text{subject to} & f_i(\vec{x}) \le 1, & i = 1, \dots, m \\ & g_i(\vec{x}) = 1, & i = 1, \dots, p \end{array}$$

In our case, $\vec{x} = \cup_{i \in \mathcal{O}}(c_i, N_i, x_i^N, x_i^{RD}, x_i^{RS})$ where $\mathcal{O}$ is the set of all operators in the query plan and $\cup$ denotes concatenation. Each operator has a vector of variables that includes: c_i, its checkpoint frequency, N_i, the number of nodes assigned to it (we assume that it is fixed in this paper), and x_i^N, x_i^{RD}, and x_i^{RS}, which capture the average time between two consecutive output tuples, requested replay tuples, and "units of recovery" (a measure of recovery speed), respectively. Tables 1 and 2 summarize these parameters (we come back to the tables shortly).

Our objective function, $f_0(\vec{x}) = T_{total}$, is the total time to execute the query including time spent recovering from failures. We define it more precisely in Sections 5.2 and 5.3.

Our constraints comprise framework and operator constraints. The former constrain how operator models are composed: (a) the average input and output rates of consecutive operators must be equal since the query plan is pipelined, (b) aggregate input and output rates for operators cannot exceed the network and processing limits, and (c) if an operator uses an output queue, it must either checkpoint its output queue to disk frequently enough, or must receive acknowledgements from downstream operators frequently enough to never run out of memory. Individual operators can add further constraints (see Section 5.3).

5.2 Objective Function

FTOpt minimizes the following cost function, that captures the expected runtime of a query plan:

$$T_{total} = \max_{p \in \mathcal{P}} \left(\mathbf{T_{P_d}} + \sum_{i=1}^{i=d-1} \mathbf{D_{P_i}} \right) + \sum_{i \in \mathcal{O}} z_i \cdot \mathbf{R_i} \tag{1}$$

The first term is the total time needed to completely process the query including the overhead of fault-tolerance if no failures occur The second term is the expected time spent in recovery from failures. Failure recovery can be added on top of normal processing because, with our approach, when a failure occurs, it blocks the entire pipeline. Indeed, even if one operator partition fails, operators upstream from that partition stop executing normally and take part in the recovery. A side-effect of this approach is that recovering a single operator partition or recovering all partitions yield approximately the same recovery time.

In more detail, for the first term, $\mathcal{P}$ is the set of all paths from the root of the query tree to the leaves. For a given path $p \in P$ of length d, the root is labeled with p_1 and the leaf with p_d; D_{p_i} is the delay introduced by operator p_i

[3]FTOpt works irrespective of a blocking operator's placement in a query plan. We focus on online query processing since FTOpt is especially useful for such plans: it enables fault-tolerance without creating unnecessary blocking.

Table 1: Functions capturing operator behavior.

Delay to produce the first tuple	
$D^N(\Theta)$	Average delay to output first tuple during normal processing (with fault-tolerance overheads).
$D^{RD}(\Theta)$	Average delay to produce first tuple requested by a downstream operator during a replay.
$D^{RS}(\Theta)$	Average delay to the start of state recovery on failure.
Average processing time	
$x^N(\Theta)$	Average time interval between successive output tuples during normal runtime (with fault-tolerance overheads).
$x^{RD}(\Theta)$	Average time interval between successive output tuples requested by a downstream operator.
$x^{RS}(\Theta)$	Average time-interval between strategy-specific "units of recovery" (*e.g.*, checkpointed tuples read from disk).
Acknowledgement interval, $a(\Theta)$, sent to upstream nodes.	

Table 2: Operator behavior parameters(Θ).

Query parameters	
$\|I_u\|$	Number of input tuples received from upstream operator u.
$\|I\|$	Number of tuples produced by current operator.
Operator parameters	
t^{cpu}	Operator cost in terms of time to process one tuple.
t^{io}	The time taken to write a tuple to disk.
Runtime parameters	
x_u^N	Average inter-tuple arrival time from upstream operator u in normal processing.
F	Fault-tolerance strategy.
c	Number of tuples processed between consecutive checkpoints.
N	The number of nodes assigned to the operator.
Surrounding fault-tolerance context	
a_d	Maximum number of unacknowledged output tuples.
x_u^{RD}	Average inter-tuple arrival time from upstream operator u during a replay.

where the delay is defined as the time taken to produce its first output tuple from the moment it receives its first input tuple; and, T_{p_d} is the time taken by the leaf operator to complete all processing after receiving its first input tuple. $T_{p_d} = D_{p_d} + x_{p_d}^N|I|$, where $|I|$ is the number of output tuples produced by the leaf operator. D_{p_i} and T_{p_d} depend on the input tuple arrival rate at an operator, which depend on how fast previous operators are in the pipeline. We capture these dependencies with constraints as mentioned in Section 5.1.

For the second term, $\mathcal{O}$ is the set of all operators in the tree. For operator $i \in \mathcal{O}$, z_i is the expected number of failures during query execution, and $\mathbf{R_i}$ is the *expected* recovery time from a failure. We estimate z_i using an administrator-provided failure model as described in Section 2.

To adapt T_{total} to be a posynomial, we need to eliminate the *max* operator. For this, we introduce a variable T for T_{total} and decompose the objective to be "minimize T with a constraint for each path such that: $(T)^{-1} \times$ [expected total time for the path] $<= 1$". Since the expected total time (for a single path) is a posynomial, the constraints are posynomials, and the entire program is a GP.

5.3 Operator Modeling

To compute the objective function, FTOpt thus requires that each operator provide expressions that characterize its delay and processing time during normal operation and when failures occur. These expressions must be provided for each fault-tolerance strategy that the operator supports. Formally, FTOpt needs to be given the functions in Table 1 expressed in terms of the parameters, represented by Θ in Table 2 (these parameters capture the inter-dependencies between operators). FTOpt combines these functions together to derive the overall processing time T_{total}.

In this section, we show how to express such functions in

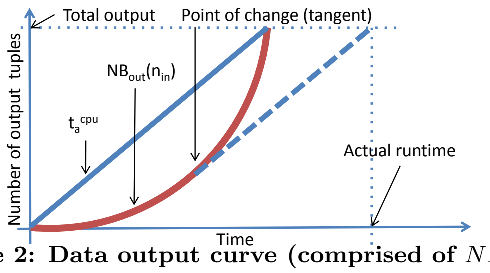

Figure 2: Data output curve (comprised of $NB_{out}(\cdot)$ curve till the "Point of change" and the dashed line after it) for a symmetric-hash join operator.

our framework. We proceed through an example: we derive the constraint equations for join (symmetric hash-join). The models for select and aggregate are similarly derived [44].

5.3.1 *Modeling Basic Operator Runtime*

To model our problem as a GP, we must (a) derive the operator output rate (given by the inter-output-tuple delay, x^N) in the absence of failures, and (b) derive the delay, D^N. The delay, however, is simply either negligible for selects and the symmetric hash-join that we model or equal to the total processing time for aggregates.[4]

The challenge in expressing an operator's output rate is that x^N can follow a complex curve for some operators such as certain non-blocking join algorithms as illustrated in Figure 2. The figure shows the data output curve for a symmetric hash-join operator. For this operator, the more tuples that it has already processed, the more likely the join is to find matching tuples, and thus it outputs tuples at an increasingly faster rate. As a result, at the beginning of the computation, the bottleneck is the input data rate (the $\mathrm{NB_{out}(n_{in})}$ curve) and the operator produces increasingly more output tuples for each input tuple. Eventually, the CPU at the join becomes the bottleneck (t_a^{cpu} curve) and the output rate flattens.

We found that ignoring such effects and assuming a constant output significantly underestimated the total runtime for the operator. Alternatively, modeling these effects and exposing them to downstream operators significantly complicated the overall optimization problem. We thus opted for the following middle-ground: we model the non-uniform output rate of an operator to derive its total runtime. Given the total runtime, we compute the equivalent average output rate that we use as a constant input arrival rate for the next operator. The GP framework is helpful here to express these non-linear behaviors.

Interestingly, we find that we can automatically derive the above curve from the following operator properties:

- t_a^{cpu}: Average time to generate one output tuple if all input is available with no delay.
- $NB_{out}(n_{in})$: This function provides the total number of output tuples produced for a given number of tuples (n_{in}) received across all input streams.

The above functions can easily be derived (hence simplifying optimizer extensibility). Both these functions are extensions of parameters of standard query optimizers: (a) t_a^{cpu} corresponds to the standard query optimizer function for computing an operator's cost, except that we then divide this cost by the operator output cardinality, and (b)

[4]To compute total query times, we ignore any partial results that an online aggregate may produce.

$NB_{out}(n_{in})$ is similar to computing the cardinality of an operator output, except that it also captures how that output is produced as the input data arrives. Simple operators like select or merge join have $NB_{out} = \sigma n_{in}$, where σ is the operator selectivity. For blocking operators such as aggregates, after the delay $D^N(\Theta)$ all the output tuples are produced at once and hence $NB_{out} = |I|$. For other non-blocking operators the relationship can be more complex as we discuss next using our symmetric hash-join as example.

For the symmetric hash-join operator, define I_{utot} to be the set of all tuples received from both upstream input channels. Hence, $|I_{utot}| = |I_1| + |I_2|$. For this operator:

$$t_a^{cpu} = |I|^{-1} \left(|I_{utot}| + |I|\right) t^{cpu}$$

The expression is a product of the average time taken to process either an input or output tuple (t^{cpu}, obtained through micro-benchmarks) and the total number of tuples seen by the operator, including the input tuples (I_{utot}) and the output join tuples (I). This number is then divided by the total number of output tuples ($|I|$) to get the average time per output tuple.

To get the NB_{out} function for a symmetric hash-join we assume that the input tuples from the two input channels can arrive in any order, each order being equally likely. Let $\hat{\sigma}$ (a function of the join selectivity σ, $|I_1|$ and $|I_2|$ [44]) be the probability that two tuples from different channels join and p_i be the probability that a tuple belongs to the $i^{th} channel$. In this case, the function $NB_{out}(n_{in})$ is defined as follows:

$$\mathrm{NB_{out}(n_{in})} = \hat{\sigma} p_1 p_2 n_{in}(n_{in} - 1) \approx \hat{\sigma} p_1 p_2 n_{in}^2$$

Intuitively, $n_{in}(n_{in}-1)$ is the count of pairs of distinct tuples to join, $p_1 p_2$ is the probability that they come from different channels, and $\hat{\sigma}$ is the probability that they join.

We now show how our optimizer translates these functions into a set of inequalities that characterize the average time interval between successive output tuples produced by an operator. For this, we require that the $NB_{out}(n_{in})$ function take the form: $NB_{out}(n_{in}) = \gamma n_{in}^k$, in order to fit into the GP framework. Thus, for our join operator: $\gamma = \hat{\sigma} p_1 p_2$ and $k = 2$. Informally, as the operator sees more input tuples, the number of the output tuples produced after processing a single new input tuple should never decrease.

Given the above, the average time interval between consecutive output tuples, x^N, is given by the following inequalities:

$$\begin{aligned} m_e &= \gamma (x^{IN})^{-k} k t_f^{k-1} \\ m_e &\leq (t_a^{cpu})^{-1} \\ m_e &\leq \gamma^{\frac{1}{k}} (x^{IN})^{-1} k |I|^{1-\frac{1}{k}} \\ (1-k^{-1}) t_f + |I| m_e^{-1} &\leq x^N |I| \end{aligned}$$

The above inequalities take x^{IN} as input, which is the time interval at which input tuples are arriving. x^{IN} depends on the current execution context. If we are operating normally, it is the average time interval between tuples produced by the upstream operators; if we are recovering from a failure, we might read the input tuples from disk at the maximum bandwidth possible for the disk.

For the exact derivation of this mode, we refer the reader to Appendix B. Here, we only provide the intuition behind it.

In the above equations, $|I|$ is the output cardinality; γ and k come from the $NB_{out}(n_{in})$ function; m_e is the number of output tuples produced per second at the instant the processing ends and t_f is the *first* time at which the output produces tuples at the rate m_e. The first equation realizes this relationship between m_e and t_f. The following inequality states that the operator can not take less than t_a^{cpu} time to produce an output tuple, since this is the least amount of time the processor needs per tuple, given the resources it has. For the second inequality, its right hand side is the maximum rate at which output could be produced if the only bottleneck was the rate of arrival of input tuples. Note that, since we require the $NB_{out}(\cdot)$ function to have a non-negative rate of change, the fastest output production rate will be at the end of the computation and the derivative of the function $NB_{out}(\cdot)$, with respect to x^{IN}, at the end gives us this value. Since, in a real computation the processing cost is positive, the actual observed rate has to be less than the derivative (the right hand side in the second inequality). The third inequality states that the total time to process all tuples (which is equal to the average output rate times the number of output produced) must be higher than the actual processing time, which is its left hand side.

To model a different operator, the functions for t_a^{cpu} and $NB_{out}(n_{in})$ would change, while the form of the inequalities and equalities used by the optimizer would remain the same. They simply use the above as parameters.

We model a partitioned operator as a single operator that scales linearly with allocated resources. This approach suffices to show the feasibility and impact of fault-tolerance optimization. We leave extensions to more complex models, including data skew between partitions, for future work.

5.3.2 Modeling Overhead of Fault-tolerance

Fault-tolerance overhead only affects t_a^{cpu}, the time an operator needs to produce an output. The model depends on the operator implementation. For MATERIALIZE, our join writes all output tuples to disk. For CHCKPT, it logs the incoming tuples to disk incrementally[5]. The join does not maintain any output queue.

For brevity, we use the notation that $\mathbb{I}^N$, $\mathbb{I}^M$ and $\mathbb{I}^C$ are 1 if NONE, MATERIALIZE or CHCKPT is the chosen fault tolerance option, respectively, and are 0 otherwise. Although we need one equation per fault-tolerance strategy we represent them as a single one.

$$t_a^{cpu} = |I|^{-1} \left(t^{cpu}(|I_{utot}| + |I|) + \mathbb{I}^C t^{io} |I_{utot}| + \mathbb{I}^M t^{io} |I| \right)$$

Here t^{io} is the time to write a tuple to disk and is also obtained through micro-benchmarks.

5.3.3 Modeling Replay Request Times

FTOpt also needs to know the average rate at which output tuples are produced to satisfy a replay request and the delay in generating the first requested tuple. The replay rate may depend on when, during the course of the query, the downstream fails. For example, if the replay behaves as during normal operations for the symmetric hash-join, it might be slower if the downstream fails early on and be faster later. To approximate the recovery rate we find the time it takes to replay all output tuples and divide that number by

[5]Out of simplicity, our join checkpoints input tuples as they arrive rather than checkpointing the hash table. When it rebuilds the hash table from a checkpoint, the operator does not redo the join.

the total number of output tuples. During this replay phase, the operator has no fault-tolerance overheads.

As before, the exact model depends on the implementation details. Our join implementation uses its in-memory hash table to regenerate outputs and hence the delay is negligible. But it could be significant for a join that can not use either its state or its output to answer tuple requests.

To get the average output rate, we reuse the framework we developed in the previous section. Thus we only need to specify t_a^{cpu} and $NB_{out}(n_{in})$ for the replay mode.

Since, during replay, we only reprocess the inputs without any fault tolerance overhead: $t_a^{cpu} = |I|^{-1} \left(|I_{utot}| + |I|\right) t^{cpu}$.

The form of the $NB_{out}(\cdot)$ remains the same as for the normal processing. Also, during reprocessing the input tuples are already in memory, hence the inter-tuple arrival time of inputs x^{IN} is at least t^{cpu} and we take $x^{IN} = t^{cpu}$.

5.3.4 Modeling Recovery Time

To compute the total time to recover from a failure, we need to know the average rate at which recovery proceeds.

As before, the exact recovery model depends on the implementation. For our join, upon failure the MATERIALIZE and the NONE options have to request all the input from the upstream nodes and rebuild the hash table exactly as it was before (using Rule 4.2 and operator determinism [44]), while CHCKPT rebuilds it from the input tuples logged to disk.

In all cases, during recovery, no output is produced when the input tuples are processed to remake the hash table. Thus, $t_a^{cpu} = t^{cpu}$ since we look at each input tuple once.

To define the function $NB_{out}(n_{in})$ we think of the hash table being rebuilt as the desired "output" and the input tuples as the inputs. Since all the input tuples are used to generate the "output" hash table: $NB_{out}(n_{in}) = n_{in}$. For MATERIALIZE and NONE, x^{IN} is the average time interval in which requested tuples from the upstream nodes arrive. For CHCKPT, since we directly read tuples from the disk: $x^{IN} = t^{io}$.

The delay in getting the first input is negligible if we use CHCKPT and is equal to the delay of the upstream tuples in the case of NONE and MATERIALIZE.

We approximate the expected hash table size to recover to be $\frac{1}{2}|I_{utot}|$. Thus, the expected time to recover is the sum of (1) the delay to receive the first input tuple, and (2) the product of the expected hash table size and the average time per tuple spent in adding a tuple to that hash table.

In summary, compared to existing cost models for parallel query runtime estimation [13, 12] and fault-tolerance in streaming engines [21], our models capture the dynamic operator interactions in pipelined queries, which we observed to affect runtime predictions and fault-tolerance optimization. For example, a fast operator following a slow one in a pipeline will produce its output slowly. At the same time, we do not require that an operator's output tuples be uniformly spread across the entire execution time of the operator [16, 49]. Indeed, because we use a GP framework, we support simple types of non-uniform outputs such as that of asymmetric hash-join. Of course, our GP framework may not cover all cases. In particular, for multi-phase operators (*e.g.*, a symmetric hash-join that spills state to disk), we may still need to split the operator into multiple sub-operators for more accurate modeling of each phase.

5.4 Approach Implementability

Our approach consists of (1) a protocol that enables heterogeneous fault-tolerance in a parallel query plan and (2) an optimizer that automatically selects the fault-tolerance strategy that each operator should use. We now discuss the difficulty of implementing this approach in a parallel data processing system.

To implement our approach, developers need to (a) implement desired fault-tolerance strategies for their operators in a manner that follows our protocol. In Section 4.2, however, we showed, how to efficiently implement three well-known fault-tolerance strategies for generic stateless and stateful operators. Existing libraries can also help with such implementation (*e.g.*, [23]). Developers must also (b) model their operator costs within a pipelined query plan. To simplify this latter task, we develop an approach that requires only that developers specify well-known functions under different fault-tolerance strategies and during recovery: an operator cost function and a function that computes how the output size of an operator grows with the input size. Our optimizer derives the resulting operator dynamics automatically. For parallel database systems [41, 14] and MapReduce-type systems such as Hive [1] or Pig [30], which come with predefined operators, the above overhead needs only be paid once and we thus posit that it is a reasonable requirement.

For user-defined operators (UDOs), the above may still be too much to ask. In that case, the simplest strategy is to treat UDOs as if they could only support the NONE or MATERIALIZE strategies (depending on the underlying platform) without ever producing acknowledgments. With this approach, UDO writers need not do any extra work at all, yet the overall query plan can still be optimized and achieve higher performance than without fault-tolerance optimization as we show in Section 6.4.

Finally, our approach relies on a set of parameters including IO cost (expressed as the time t^{io} spent in a byte sized disk IO), per-operator CPU cost (expressed as the time t^{cpu} spent processing each tuple), and total network bandwidth. Commercial database systems already automate the collection of such statistics (*e.g.*, [31]), though t^{cpu} is typically expanded into a more detailed formula.

Other necessary information includes the expected number of failures for the query (see Section 2), operator selectivities (standard optimizer-provided metric), and an estimate of the total checkpointable state. As shown in Section 6.6, our optimizer is insensitive to small errors in these estimates.

Overall, the requirements of our fault-tolerance optimization framework are thus similar to those of existing cost-based query optimizers.

5.5 Handling Complex Failure Scenarios

So far, we have focused on process failures. However, our approach also handles other types of failures.

Our approach still works when entire physical machines fail (*e.g.*, due to a disk failure, a power failure, or a network failure). To support such failures, checkpoints must be written to remote nodes instead of locally [19], which adds network and CPU costs that must be taken into account by the optimizer. Given that the optimizer knows the size of these checkpoints, it can take that cost into account. Second, when a physical machine fails or becomes disconnected, the number of nodes in the cluster is reduced by one, which must also be taken into account by the optimizer.

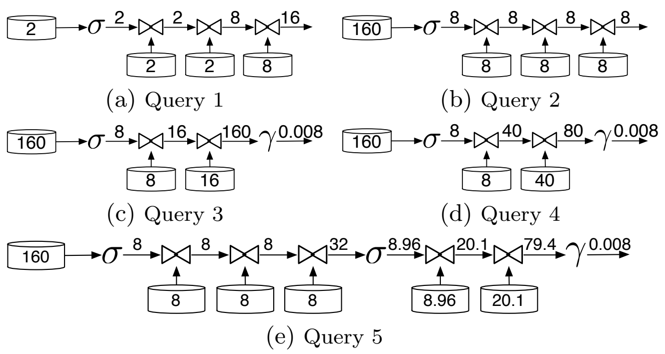

Figure 3: Query plans used in experiments. σ, $\bowtie$, γ denote Select, Join, and Aggregation, respectively. All numbers are in millions.

Our approach does not currently handle failures that affect a large number of machines. Indeed, such failures can cause the temporary loss of input data or checkpointed data. In such cases, the query needs to be restarted in its entirety once the input data becomes available again. In general, however, large-scale rack and network failures are infrequent, while single machine failures are common. For example, Google reports 5 average worker deaths per MapReduce job in March 2006 [9], but only approximately 20 rack failures per year (and similarly few network failures) [8].

Even though our approach does not handle large-scale failures that cause the loss of input or checkpointed data, it does handle multiple operators failing at the same time. The only requirement in such cases is that operators be restarted from downstream to upstream, ensuring that each operator knows where to start recovering from before asking upstream neighbors to replay data.

6. EVALUATION

We evaluate FTOpt by answering the following questions: (1) Does the choice of fault-tolerance strategy for a parallel query matter? (2) Are there configurations where a *hybrid* plan, where different operators use different fault-tolerance techniques, outperforms uniform plans? (3) Is our optimizer able to find good fault-tolerance plans automatically? (4) How do user-defined operators affect FTOpt? (5) What is the scalability of our approach? (6) How sensitive is FTOpt to estimation errors in its various parameters?

We answer these questions through experiments with a variety of queries in a 17-node cluster. Each node has dual 2.5 GHz Quad Core E5420 processors and 16 GB RAM running Linux kernel 2.6.18 with two 7.2K RPM 750 GB SATA hard disks. The cluster runs a simple parallel data processing engine that we wrote in Java. The implementation includes our new fault-tolerance framework and specific per-operator fault-tolerance strategies for a small set of representative operators. All fault-tolerance strategies were moderately optimized (see Section 4.2). We implemented the optimizer in MATLAB using the *cvx* package [7].

The query plans that we use in the experiments are shown in Figure 3. They include an SJJJ and SJJA query (we also test a more complex query later in this section). For both queries we have 8 partitions per operator with 2 cores and 1 disk per partition. Partitions of the same operator run on different machines. The input data is synthetic and without skew. Tuples are 0.5 KB in size. The schema consists of 4

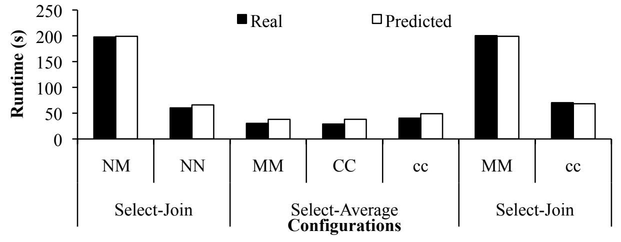

Figure 4: Runtime without failures for various two-operator queries. X-axis labels show the fault-tolerance strategy chosen: N for NONE, M for MATERIALIZE, C for CHCKPT with a total of 10 checkpoints, and c for CHCKPT with 1K checkpoints.

attributes used to hash-partition tuples for each operator, a 5th attribute for grouping the aggregates, and a 6th one for the join predicates. A separate producer process generates input tuples. For a given plan, we get the expected recovery time by injecting a failure midway through the time the plan takes to execute with no failures. We inject *exactly one* failure per run and show the recovery time averaged over all distinct operators in the plan.

6.1 Model Validation Experiments

FTOpt requires the t^{cpu} and the t^{io} values for each operator. It also requires the network bandwidth for each machine in the cluster. Through micro-benchmarks, we find that the average time to read a tuple from disk (sequential read) is $t^{io} = 13.0$ μs for a 0.5 KB tuple. This number is equivalent to a disk throughput of 37 MBps. For select and aggregate operators, we measure t^{cpu} to be $1.82\mu s$. The join operator, internally, works in two parts: (1) hashing the input tuple and storing it in one of the tables for a cost of $t_1 = 8\mu s$ and (2) joining the hashed input tuple to the corresponding tuples from the other table for a cost of $t_2 = 1\mu s$. We use t_1, t_2, and the operator's selectivity to estimate its t^{cpu}. Finally, we measure the network I/O time per 0.5 KB tuple to be $4.7\mu s$, which is equivalent to a network bandwidth of 109.4 MBps and is close to the theoretical maximum of 1 Gbps network bandwidth for each machine in the cluster.

These parameters along with our operator models enable FTOpt to predict the runtime for an entire query plan. Figure 4 shows the runtime without failure for a few two-operator queries. While the median percentage difference between real and predicted runtime is 9.5%, this error is small given the overall differences in runtime between various configurations. We measure the sensitivity of our approach to the benchmarked parameter values in Section 6.6.

6.2 Impact of Fault-Tolerance Strategy

The first question that we ask is whether a fault-tolerance optimizer is useful: how much does it really matter what fault-tolerance strategy is used for a query plan?

Figures 5 through 7 show the actual and predicted runtimes for Queries 1 through 3 from Figure 3 with 8 partitions per operator. Note that, each join receives input from *two* sources: its upstream operator in the plan and a producer process. In all our experiments, an equal number of tuples was received from each source. Whenever FTOpt selects CHCKPT as a strategy, it also chooses the checkpoint frequency (Query 3). In other cases, we use 100 checkpoints, a

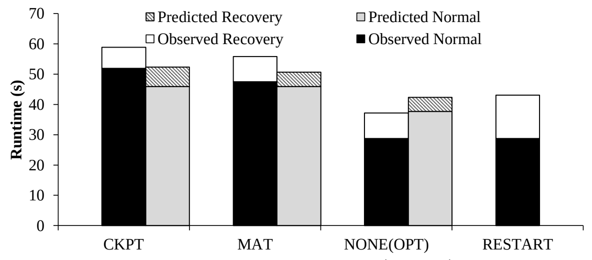

Figure 5: Query 1 (SJJJ)

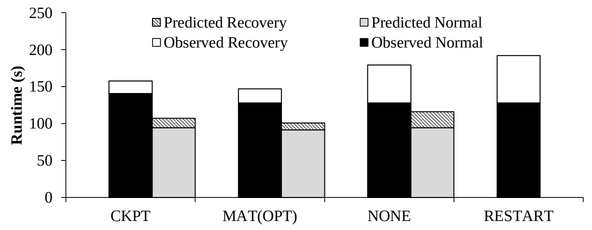

Figure 6: Query 2 (SJJJ with lower selectivities)

manually selected value that we found to give high performance in these experiments.

The most important result from these experiments is that, while these queries are all similar to each other, *each one requires a different fault-tolerance plan to achieve best performance.* For Query 1, a uniform NONE strategy is best. For Query 2, uniform MATERIALIZE wins. Finally, for Query 3, uniform CHCKPT outperforms the other options.

Second, restarting a query is at most 50% slower than a strategy with more fine-grained fault-tolerance. The fine-grained strategy gains the most when it reduces recovery times with minimal impact on runtime without failures. For some queries, the appropriate choice of fault-tolerance gets close to this theoretical upper bound. For Query 2, RESTART is 31% worse than the best strategy while for Query 3, restarting is 44% slower than the best strategy. Achieving such gains, however, requires fault-tolerance optimization. Indeed, different strategies win for different queries and a wrong fault-tolerance strategy choice leads to much worse performance than restarting a query. Overall, the differences between the best and worst plan are high: 58% for Query 1, 31% for Query 2, and 72% for Query 3.

Finally, in all cases, *FTOpt is able to identify the winning strategy!* Predicted runtimes do not always match the observed ones exactly. Most of the difference is attributable to our simple model for the network and FTOpt's predictions are thus more accurate when either CPU or disk IO is the bottleneck in a query plan and less accurate when it is the network. While we could further refine our models, to pick the optimal strategy, we only need to have correct relative order of predicted runtimes for different plans. As shown in Figures 4 through 8, FTOpt preserves that order when runtime differences are large. When two configurations lead to very similar runtimes, FTOpt may not find the best of these plans but the choice of plan matters less in such a case and FTOpt always suggests one of the good plans.

In summary, the correct choice of fault-tolerance strategy can significantly impact query runtime and that choice is not obvious as similar query plans may require very different strategies. FTOpt can automatically select a good plan.

6.3 Benefits of Hybrid Configurations

We now consider a query (Query 4), similar to Query

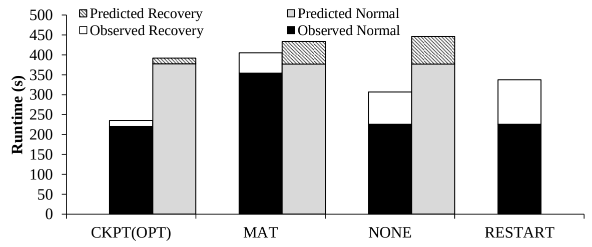

Figure 7: Query 3 (SJJA query)

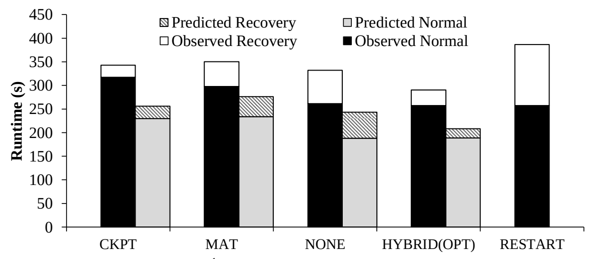

Figure 8: Query 4 (SJJA with more expensive joins). The hybrid strategy is to materialize after select, do nothing for joins, and checkpoint the aggregate

3, but with the joins processing and producing much more data, making checkpointing expensive. Figure 8 shows that a hybrid strategy that materializes the select's output, does nothing for the joins, and checkpoints the aggregate's state for a total of 40 checkpoints (value selected by the optimizer), yields the best performance. The uniform strategies are 15% slower at best and 21% slower at worst while RESTART is 35% slower.

We observe similar gains for a longer query (Query 5) with *eight* operators. Figure 9 shows that the hybrid plan (chosen by the optimizer) materializes both selects' outputs, does nothing for joins and takes 20 checkpoints of the aggregate. The best and worst uniform strategies and RESTART are 16%, 23% and 36% slower, respectively. Manually, we found that checkpointing the first two joins in the hybrid plan led to another hybrid plan that was 2% faster. While the optimizer did not choose this better plan, the plan it chose performs similarly. Further, both the observed and predicted best plans are hybrid.

The experiments thus show that hybrid fault-tolerance strategies can be advantageous and the best strategy for an operator depends not only on the operator but on the whole query plan: the same operator can use different strategies in different query plans: *e.g.*, select in Queries 3 and 4.

Note that we inject only one failure per experiment. Thus, our graphs show the minimum guaranteed gains. Additional failures amplify differences between strategies.

6.4 Performance in Presence of UDOs

We look at the applicability of heterogeneous fault-tolerance when an operator is a user-defined function with limited fault-tolerance capabilities. We experiment with Query 3, but treat its last operator, the aggregate, as a UDO that can only restart from scratch if it fails. Note that Rule 4.2 and operator determinism [44] allow restarting a UDO in isolation without restarting the entire query.

Figure 10 shows the results. Previously, the best fault-tolerance strategy, with a single failure, was to checkpoint every operator ("With CKPT") and checkpointing aggregate provided significant savings in recovery time. Now that the aggregate can use NONE as sole strategy, we find that ma-

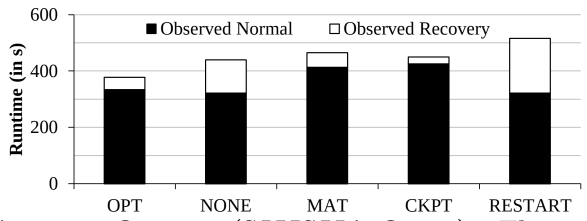

Figure 9: Query 5 (SJJJSJJA Query). The optimal hybrid strategy is MNNNMNNC where M denotes MATERIALIZE and N denotes NONE and C denotes CHCKPT. In the optimal configuration 20 checkpoints are taken.

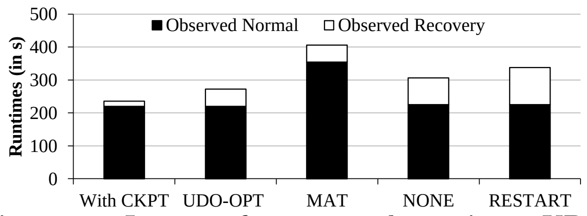

Figure 10: Impact of aggregate becoming a UDOs without fault-tolerance capabilities on Query 3. The optimal strategy is to materialize after select and do nothing elsewhere.

terializing the first operator's output and using NONE for the remaining operators outperforms uniformly materializing, none and RESTART by 48%, 12%, and 24%, respectively. The hybrid strategy is itself 16% slower than the optimal strategy for Query 3 ("With CKPT").

Hence even in the presence of fault-tolerance agnostic UDOs, FTOpt can generate significant runtime savings.

6.5 Scalability

FTOpt currently uses a brute force search algorithm, but we find that simple heuristics can significantly prune the search space. Indeed, we observe that the best hybrid plans use the NONE strategy for many operators and using another strategy in place of NONE will always increase the runtime *without failures*. Thus, if the runtime without failure for a plan exceeds the runtime with failures for another plan, we can prune the former plan. Hence, evaluating plans in the decreasing order of the number of operators that use the NONE strategy can prune significant fractions of the search space. For example, with this heuristic, the optimizer examines only 28 out of 81 configurations for Query 4. In addition, the search essentially computes the least costly of a set of independent optimization problems and all of these problems can be optimized in parallel.

FTOpt's MATLAB implementation uses the cvx package, which offers a *successive approximation* solver using SDPT3 [42]. In our prototype, the average time to solve the optimization problem per plan is around 25s for the 4 operator plans in the previous sections. However, an optimized solver can solve a larger problem in *a sub millisecond* [29]. The behavior of an operator for a fault-tolerance strategy is modeled using at most 12 inequality and 4 equality constraints of 11 variables. Thus, a query with n operators can be modeled using $11n$ variables, $13n + 1$ inequality and $4n$ equality constraints. Further, all but one of the constraints are sparse: they depend on just a few variables independent of n. For example, with 4 operators, our models use 44 variables, 16 equalities, and 53 inequalities. The existing optimized solvers can solve a problem of 140 variables, 120 equalities, and 60 inequalities in 0.425 ms on average [29].

To sum up, with an optimized solver, and a parallelized heuristic search algorithm, FTOpt could be scalable enough to handle larger query plans within a few seconds.

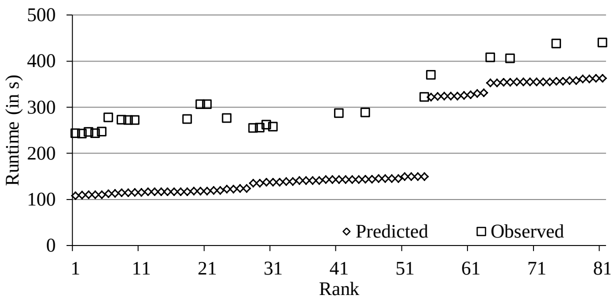

Figure 11: Observed and predicted runtimes for Query 3, sorted on the predicted runtime, for all 81 fault-tolerance plans for the query.

Table 3: Real rankings of top 5 plans from perturbed configurations.

Perturbation	**Rankings**				
Failing thrice instead of once	1	2	3	4	5
IO cost 2.0x of true value	1	6	8	9	18
IO cost 0.5x of true value	2	1	3	4	5
IO cost 10x of true value	6	18	20	21	24
IO cost 0.1x of true value	2	28	31	30	29
Selectivity of all operators 1.1x	1	2	3	4	5
Selectivity of all operators 0.9x	1	2	3	4	5
Selectivity of all operators 2.0x	1	2	3	4	5
Selectivity of all operators 0.5x	56	1	66	67	10

6.6 Optimizer Sensitivity

We evaluate FTOpt's sensitivity to inaccuracies in parameter estimates. We experiment with Query 3 since it is most sensitive to wrong choices: Figure 11 shows that runtimes vary from about 250s to 400s depending on the chosen plan.

To evaluate the sensitivity for a given parameter, we rerun FTOpt, feeding it a perturbed parameter value. We only perturb a single parameter at a time while keeping the other parameters at their true values. We then compute the top 5 plans with the perturbed value and report the ranks of these plans in FTOpt's original ranking (Figure 11). Table 3 shows the results. As an example, in this table, when IO cost increases to 2X its true value, the second best plan identified by FTOpt was ranked 6th with the real IO costs.

Table 3 shows that FTOpt is very robust to small errors in the number of failures and it is fairly robust to even large errors in IO cost: a 10x change still leads to a good plan (with true rank 6) being chosen, though the subsequent plans have poor true rankings. FTOpt is least robust to cardinality estimation errors. In our experiments, we varied the selectivities of all the operators in tandem (and with the join always processing the same number of tuples from both streams). In this scenario, our predictions were unchanged for changes of 1.1x, 2x and 0.9x in selectivity but for a 0.5x change, the top choice's true rank was 56 with an observed runtime about 70% worse than that of the best configuration possible.

The robustness to I/O cost errors and failure errors can be explained by the fact that the effect of these errors is mostly linear on the optimizer. However, imprecise selectivity estimates have an exponential effect (the further an operator is from the beginning, the less data it processes and it produces even less output) on FTOpt. Thus, the optimizer is more sensitive to perturbations in selectivity estimates.

7. CONCLUSION

In this paper, we presented a framework for heterogeneous fault-tolerance, a concrete instance of that framework, and FTOpt, a latency and fault-tolerance optimizer for parallel data processing systems. Given a pipelined query plan, a shared-nothing cluster, and a failure model, FTOpt selects the fault-tolerance strategy for *each* operator in a query plan to minimize the time to complete the query with failures. We implemented our approach in a prototype parallel query processing engine. Our experimental results show that different fault-tolerance strategies, often hybrid ones, lead to the best performance in different settings and that our optimizer is able to correctly identify a winning strategy.

Acknowledgments

We thank Phil Bernstein, Bill Howe, Julie Letchner, Dan Suciu, and the anonymous reviewers for helpful comments on the paper's early drafts. This work is supported in part by the National Science Foundation grants NSF CAREER IIS-0845397 and IIS-0713123, gifts from Microsoft Research, and Balazinska's Microsoft Research Faculty Fellowship.

8. REFERENCES

[1] Ashish Thusoo et. al. Hive - a petabyte scale data warehouse using hadoop. In *Proc. of the 26th ICDE Conf.*, 2010.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the SIGMOD Conf.*, June 2005.

[3] S. P. Boyd, S. J. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. Technical report, Stanford University, Info. Systems Laboratory, Dept. Elect. Eng., 2004.

[4] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proc. of the SIGMOD Conf.*, 2007.

[5] S. Chaudhuri, R. Kaushik, A. Pol, and R. Ramamurthy. Stop-and-restart style execution for long running decision support queries. In *Proc. of the 33rd VLDB Conf.*, 2007.

[6] Chen et. al. High availability and scalability guide for DB2 on linux, unix, and windows. IBM Redbooks `http://www.redbooks.ibm.com/redbooks/pdfs/sg247363.pdf`, Sept. 2007.

[7] Cvx. `http://www.stanford.edu/~boyd/cvx/`.

[8] J. Dean. Software engineering advice from building large-scale distributed systems. `http://research.google.com/people/jeff/stanford-295-talk.pdf`.

[9] J. Dean. Experiences with MapReduce, an abstraction for large-scale computation. Keynote I: PACT, 2006.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), 2002.

[12] S. Ganguly, A. Goel, and A. Silberschatz. Efficient and accurate cost models for parallel query optimization (extended abstract). In *Proc. of the 15rd PODS Symp.*, 1996.

[13] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the SIGMOD Conf.*, pages 9–18, 1992.

[14] Greenplum database. `http://www.greenplum.com/`.

[15] Hadoop. `http://hadoop.apache.org/`.

[16] W. Hasan and R. Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proc. of the 20th VLDB Conf.*, 1994.

[17] J. M. Hellerstein, R. Avnur, and V. Raman. Informix under CONTROL: Online query processing. *Data Mining and Knowledge Discovery*, 4(4):281–314, 2000.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.

[19] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. of ICDE Conf.*, Apr. 2007.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the EuroSys Conf.*, pages 59–72, 2007.

[21] Jeong-Hyon Hwang et. al. High-availability algorithms for distributed stream processing. In *Proc. of the 21st ICDE Conf.*, Apr. 2005.

[22] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. of the 1st ACM symposium on Cloud computing (SOCC)*, pages 181–192, 2010.

[23] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. In *Proc. of the 34th VLDB Conf.*, 2008.

[24] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. *SIGMOD Record*, 29(2):46–57, 2000.

[25] A.-P. Liedes and A. Wolski. Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Proc. of the 22nd ICDE Conf.*, page 99, 2006.

[26] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proc. of the 1st ACM symposium on Cloud computing (SOCC)*, 2010.

[27] D. Lomet. Dependability, abstraction, and programming. In *DASFAA '09: Proc. of the 14th Int. Conf. on Database Systems for Advanced Applications*, pages 1–21, 2009.

[28] Marcos Vaz Salles et. al. An evaluation of checkpoint recovery for massively multiplayer online games. In *Proc. of the 35th VLDB Conf.*, 2009.

[29] J. Mattingley and S. Boyd. Automatic code generation for real-time convex optimization. In *Convex Optimization in Signal Processing Optimization*. Cambridge U. Press, 2009.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of the SIGMOD Conf.*, pages 1099–1110, 2008.

[31] Oracle database. `http://www.oracle.com/`.

[32] A. Pavlo et. al. A comparison of approaches to large-scale data analysis. In *Proc. of the SIGMOD Conf.*, 2009.

[33] L. Raschid and S. Y. W. Su. A parallel processing strategy for evaluating recursive queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 412–419. Morgan Kaufmann, 1986.

[34] A. Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. An Oracle white paper, Mar. 2002.

[35] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Proc. of the 5th ICDE Conf.*, pages 452–462, 1989.

[36] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[37] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the SIGMOD Conf.*, June 2004.

[38] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing etl workflows for fault-tolerance. In *Proc. of the 26th ICDE Conf.*, 2010.

[39] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the 23rd PODS Symp.*, June 2004.

[40] R. Talmage. Database mirroring in SQL Server 2005. `http://www.microsoft.com/technet/prodtechnol/sql/2005/dbmirror.mspx`, Apr. 2005.

[41] Teradata. `http://www.teradata.com/`.

[42] R. H. Tütüncü, K. C. Toh, and M. J. Todd. Solving semidefinite-quadratic-linear programs using SDPT3. *Mathematical programming*, 95(2):189–217, 2003.

[43] Tyson Condie et. al. MapReduce online. In *Proc. of the 7th NSDI Symp.*, 2010.

[44] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. Technical report, Department of Computer Science and Engineering, Univ. of Washington, 2010.

[45] Vertica, inc. `http://www.vertica.com/`.

[46] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the*

First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991, pages 68–77. IEEE Computer Society, 1991.

[47] C. Yang, C. Yen, C. Tan, and S. R. Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proc. of the 26th ICDE Conf.*, 2010.

[48] Yuan Yu et. al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of the 8th OSDI Symp.*, 2008.

[49] M. Zaït, D. Florescu, and P. Valduriez. Benchmarking the DBS3 parallel query optimizer. *IEEE Parallel Distrib. Technol.*, 4(2):26–40, 1996.

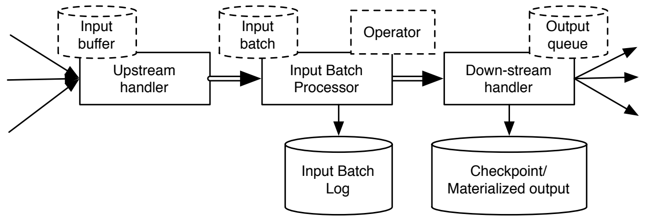

Figure 12: Architecture of the operator framework. The operator processes the incoming data in a pipelined manner. Threads are assigned to each stage of the pipeline and thus each stage runs concurrently. Network IO is handled by a pool of threads. The dashed components represent inmemory data structures and the implementation of (user-supplied) operator logic.

APPENDIX

In this Appendix, we provide additional information about various aspects of our framework and the FTOpt optimizer.

A. IMPLEMENTATION

The prototype is written in Java and built on top of the Apache MINA framework (`http://mina.apache.org/`) to implement efficient network IO. The current implementation can run a directed acyclic graph (DAG) of operators. At runtime, each operator in the DAG is replicated across multiple machines and executed in parallel. The data communication between upstream and downstream operators is done using all-to-all TCP connections.

A.1 Operator Framework Architecture

Figure 12 illustrates the framework of an operator. The framework has three concurrently executing components: *Upstream Handler*, *Batch Processor*, and *Downstream Handler*. The three components make up a data processing pipeline connected by queues. We now walk through how the incoming data is processed by this pipeline.

First, for each upstream partition, the Upstream Handler buffers the input tuples and creates a batch of input tuples whenever there are enough tuples or when the stream is stalled (which is detected by a timeout.) Both the size of an input tuple batch and the timeout are configurable parameters.

Next, the input batch is handed to the Input Batch Processor (IBP). Before running the core operator algorithm, the batch processor logs the summary information for the current batch for deterministic replay of the input stream. Because the tuples in an input tuple batch are all from the same upstream partition, in the batch summary, we only need to record the upstream partition identifier, the first tuple identifier in the batch, and the number of tuples in the batch for deterministic replay. In Appendix A.4, we discuss the details of logging and show that logging imposes a minimal overhead. The output of core operator algorithm is also collected in a batch and handed to the Downstream Handler.

Finally, the Downstream Handler streams the output tuples to the downstream operators and completes the processing of a batch of input tuples. The output tuples are routed to designated downstream operators according to a partition function. The downstream handler also takes the required fault-tolerance action such as materializing output before writing to the network or triggering checkpoints to capture the current state of the operator at the end of processing the current output batch.

Our prototype supports the three fault-tolerance strategies we mention in Section 4.2: NONE, MATERIALIZE, and CHCKPT. CHCKPT is supported only when the operator algorithm implements necessary hooks (serialize and de-serialize state.) The other two strategies are supported automatically by the framework. For stateless operator such as *Select* and *Project*, when the strategy NONE is chosen, the prototype supports skipping over the previously processed input during recovery and replay.

A.2 Operator Implementation

The current prototype implements three representative relational operators: *Select*, *Aggregate*, and *Join*. For fault-tolerance strategy, we only describe the detail of CHCKPT because NONE and MATERIALIZE strategies are automatically supported by the framework.

Select: This operator evaluates a given predicate on each input tuple. We did not implement checkpoint hooks for it since it is stateless.

Aggregate: This operator computes the average of a specific column in a group. It keeps track of the partially aggregated states using an in-memory hash table. We implemented checkpoint hooks to store the in-memory hash table into a checkpoint and load it from a checkpoint.

Join: We implemented a binary symmetric hash join operator using two in-memory hash tables [33, 46]. We implemented incremental checkpoints: input tuples are buffered in memory, written to disk when a checkpoint occurs, and then deleted from memory. During recovery, the operator rebuilds its hash tables by reading input tuples from the disk. No joins need to be performed at this point. During replay, the operator first locates the oldest input tuple to replay, then joins the following input tuples with the in-memory state.

A.3 Synthetic Benchmark Setup

We implemented the synthetic workload in Section 6 as follows:

- **Data:** All fields are randomly generated integers or strings.
- **Partition:** We hash-partition the output of each operator on a different attribute. We use the randomly generated value for that attribute to determine where to route each tuple.
- **Select:** We send a tuple to the output if the random value of the select field -of type double and taking values from 0 through 1- is less than the given selectivity.
- **Aggregate:** We vary the state size by controlling the number of groups to which the input tuples are aggregated.
- **Join:** Given the join selectivity σ, we join two input tuples when the join attribute, for the two tuples, is congruent modulo $\lceil \sigma^{-1} \rceil$.

A.4 Ensuring Operator Determinism

Our framework requires that the operator partitions be deterministic. In particular, rule 4.2 requires that, in response to a valid request, a partition must always return the same sequence of tuples, irrespective of any failures it experiences.

Most relational operators (and hence their partitions) can be made deterministic as long as when they restart, they process the same tuples in the same order across all their inputs. The challenge is that these inputs come from different machines in the cluster and may thus arrive with different latencies when they are replayed. One approach to ensure a deterministic input-data order is to buffer and interleave tuples using a pre-defined rule [2, 39]. These techniques, however, can impose significant memory and latency overheads due to tuple buffering.

Instead, we adopt the approach of logging determinants [11]. As the operator receives input tuples, it accumulates them into small batches, with one batch per input relation partition. For example, consider an operator with two inputs coming from parent operators p^1 and p^2. Tuples arrive on these inputs starting with tuple id p^1_1 from p^1 and p^2_1 from p^2. Tuples arrive in interleaved order and the operator accumulates them into batches, buffering these two batches separately in memory while maintaining the tuple arrival order within a batch. Whenever a particular batch exceeds a pre-defined size or receives an end-of-stream signal, the operator writes a log entry to disk that contains: the identifier of the stream for this batch, the identifier of the first tuple in the batch, and the number of tuples in the batch. Each log entry also has an implicit log sequence number (lsn) that is not written to disk. The logging is done *before* processing a batch. The operator processes the batches in the same order in which it writes their log entries to disk. In our example, if we use a batch size of 2500, and the operator receives 3500 tuples from p^1 and 4000 tuples from p^2, the logged entries might look as follows: $\langle 2, p^2_1, 2500\rangle, \langle 1, p^1_1, 2500\rangle, \langle 1, p^1_{2501}, 1000\rangle, \langle 2, p^2_{2501}, 1500\rangle$.

Log entries are force-written to stable storage but, as we show below, this logging overhead is negligible even for batch sizes as small as 512 tuples per batch. If the operator needs to reprocess its input, it uses the log to ensure the reprocessing occurs in the same order as before. To avoid expensive disk IOs when possible (*i.e.*, when the operator itself does not fail but its downstream neighbor fails), recent determinants are cached in memory.

Before processing an input tuple, the operator tags it with $\langle lsn, psn\rangle$, where lsn corresponds to the log entry sequence number of the corresponding batch and psn is the tuple order within that batch. This information is used to assign unique tuple identifiers to output tuples. Note that all log entries are of a constant size and a lsn is enough to index a log entry.

Output tuple identifiers consist of three integer fields: $\langle lsn, psn, seq\rangle$. The first two fields identify one of the input tuples that contributed to this output tuple. A sequence number, seq, is added since one input tuple can contribute to many output tuples (as in the case of joins.)

As an example, we show how we use this mechanism to generate unique identifiers for tuples produced by the following operators:

- Select: Our select always has a selectivity less than or equal to one and can thus propagate the input tuple identifier onto the output tuple, setting seq to zero.
- Join: The latest tuple that led to the creation of this tuple is used to populate the first two fields. The third field is a count of the number of matches for any given tuple.
- Aggregate: Since aggregates are blocking operators, they do not need a log. In case we use CHCKPT, we can store the last tuple identifiers received from each of the upstream partitions when we make the checkpoint.

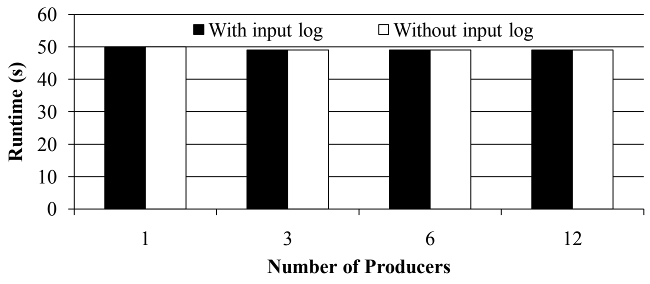

Figure 13: Each pair of bars represents the time to complete processing, with and without logs, with a different number of upstream producers for a select operator. There is virtually no overhead even for 12 input streams.

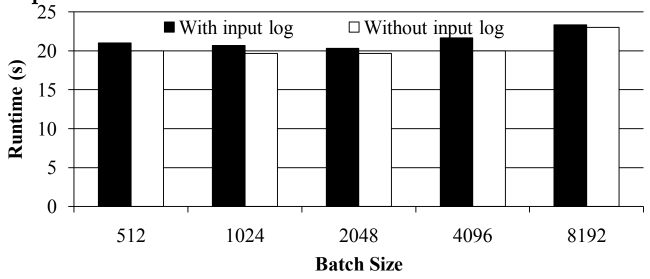

Figure 14: Each pair of bars represents the time to complete processing, with and without logs, with different batch sizes for a join operator. The minimum overhead occurs with a batch size of 2048.

To validate that logging overhead is negligible, we execute a select operator on a single machine with an input of size 2.5×10^6 tuples (or 1.19 GB) and we vary the number of upstream producers while keeping the batch size fixed at 512 tuples. Figure 13 shows the time to process all tuples with and without logging enabled. The results show that the logging mechanism scales well with the number of upstream producers. The average runtimes of three runs rounded to the nearest second are identical.

To select the optimal log batch size we execute a join operator that processes 1 million tuples from each of its two inputs. It is a 1x1 foreign key join and produces 1 million output tuples. We have a total of four producers generating all the data and we vary the log batch size from 512 to 8192. As Figure 14 shows, the smallest runtime overhead was 3% for a batch size of 2048 tuples. As expected the runtime with no logs for smaller batch sizes remains the same as that for 2048 while the runtime with logging increases since we write more log entries if batch sizes are smaller and more cpu time is spent in writing the log entries to disk. It should be noted that the runtime with and without logs increases for batch sizes of 4096 and 8192. This is because of an increased buffering delay for each input batch. In all our experiments, we use a batch size of 2048 and a tuple size of 0.5 KB.

A.5 Resource Allocation in FTOpt

In addition to fault-tolerance strategies, FTOpt also produces an allocation of resources to operators because resource allocation and choice of fault-tolerance strategy are tightly interconnected. Resource allocation is expressed as a fraction of all available CPU and network bandwidth resources. Bandwidth is further constrained by network topology.

In this paper, we make several simplifying assumptions to implement and test our proof-of-concept optimizer. We assume a simple setting where the set of compute nodes are connected through a single switch. The current version of our optimizer abstracts out the resource allocation by assuming that the time to process each tuple and the disk IO costs scale linearly with the amount of resources allocated to an operator. Thus, if a single operator partition of operator i takes t^{cpu} time to process a tuple, then with n_i partitions each assigned exclusively to a machine, the effective time the operator (*i.e.*, all the operator partitions together) takes to process a single tuple is $\frac{t^{cpu}}{n_i}$ time. Similarly the time taken to write a tuple to disk is taken to be $\frac{t^{io}}{n_i}$. Our optimizer handles fractional resource assignments.

Given a resource allocation, operators can either be co-scheduled on the same physical nodes (*i.e.*, all nodes execute all operators) or separated onto different nodes (*i.e.*, each node executes a subset of all operators.) In the latter case, resource allocation must be rounded-off to the granularity of machines, which can lead to lower performance. In the former case, operators may end-up writing their logs and checkpoints to the same disks for a more complex performance curve for these interleaved IO operations. While our optimizer handles both strategies and computes fractional resource assignments, in our experiments, we pinned each operator partition to its own core and its own disk on each node to keep our models simple.

B. OPERATOR MODELING STRATEGY

We refer the reader to Section 5.3.1 for an overview of what information about each operator in the pipeline we require so as to automatically infer the runtimes for the entire pipeline. To recap: We only require (a) the expected number of output tuples produced given the number of input tuples received, and (b) the average cpu time required to produce each output tuple.

We remind the reader that we have abstracted away the different number of partitions of each operator by dividing the time to process each tuple and the time to read or write each tuple by the number of machines allocated to the operator.

We now restate the equations and explain how the equations are derived.

$$\begin{aligned} m_e &= \gamma(x^{IN})^{-k} k t_f^{k-1} &(2)\\ m_e &\le (t_a^{cpu})^{-1} &(3)\\ m_e &\le \gamma^{\frac{1}{k}}(x^{IN})^{-1} k |I|^{1-\frac{1}{k}} &(4)\\ (1-k^{-1})t_f + |I| m_e^{-1} &\le x^N |I| &(5)\end{aligned}$$

In the above equations, $|I|$ is the output cardinality; γ and k come from the $NB_{out}(n_{in})$ function; m_e is the number of output tuples produced per second at the instant the processing ends and t_f is the *first* time at which the output produces tuples at the rate m_e.

Equality 2 realizes this relationship between m_e and t_f. Specifically, the rate at which output tuples are produced by the operator after time t_f is

$$\begin{aligned} m_e &= \frac{d}{dt} NB_{out}(n_{in})\Big|_{t=t_f}\\ &= \frac{d}{dt}\gamma n_{in}^k\Big|_{t=t_f}\\ &= \frac{d}{dt}\gamma\left(\frac{t}{x^{IN}}\right)^k\Big|_{t=t_f}\\ &= \gamma(x^{IN})^{-k} k t_f^{k-1}\end{aligned}$$

Inequality 3, $m_e \le (t_a^{cpu})^{-1}$, states that the operator can not take less than t_a^{cpu} time to produce an output tuple, since this is the least amount of time the processor needs per tuple, given the resources it has. The inequality becomes an equality when the operator reaches a stage in its processing when the processor is working at its full capacity and can not keep up with the theoretically maximum rate at which output tuples could be produced given the rate at which the input tuples are being received.

For inequality 4, $m_e \le \gamma^{\frac{1}{k}}(x^{IN})^{-1}k|I|^{1-\frac{1}{k}}$, its right hand side is the maximum rate at which output could be produced if the only bottleneck was the rate of arrival of input tuples. Note that, since we require the $NB_{out}(\cdot)$ function to have a non-negative rate of change, *i.e.*, the fastest output production rate will be at the end of the computation and the derivative of the function $NB_{out}(\cdot)$ at the end gives us this value. Formally,

$$\begin{aligned} rhs &= \frac{d}{dt}\gamma\left(\frac{t}{x^{IN}}\right)^k\Big|_{t=t_{end}}\\ &= \gamma\frac{k}{x^{IN}}\left(\frac{t_{end}}{x^{IN}}\right)^{k-1}\\ \text{where, } |I| &= \gamma\left(\frac{t_{end}}{x^{IN}}\right)^k \Rightarrow \frac{t_{end}}{x^{IN}} = \left(\frac{|I|}{\gamma}\right)^{\frac{1}{k}}\\ \Rightarrow rhs &= \gamma^{\frac{1}{k}}(x^{IN})^{-1}k|I|^{1-\frac{1}{k}} \text{ (By substitution.)}\end{aligned}$$

Since, in a real computation the processing cost is positive, the actual observed rate has to be less than the derivative (the right hand side in the second inequality.)

The last inequality states that the total time to process all tuples (which is equal to the average output rate times the number of output produced and appears on the right-hand side of the inequality) must be higher than the actual processing time, which is its left hand side. Let S be the number of output tuples produced until the operator was bound by the input arrival rate. After this point, either all the output tuples have been produced or the subsequent processing is limited by the compute resources. In other words, S is the number of output tuples produced until time t_f. Specifically,

$$t_f + m_e^{-1}(|I| - S) \le x^N|I|$$

where

$$S = \gamma\left(\frac{t_f}{x^{IN}}\right)^k \qquad (6)$$

In the inequality above the first term is the time spent in the input-limited stage of operation while the second term is the time spent in the compute-limited stage of the operator's operation. Substituting the value of S from Equation 6 and the value of m_e^{-1} from Equation 2 we get Equation 5.

The analysis above yields a form suitable for geometric programs as long as t_a^{cpu} is a posynomial and $NB_{out}(n_{in}) = \gamma n_{in}^k$ for a constant $k \geq 1$ and γ being a monomial.

B.1 First Tuple Delay

The delay to produce the first tuple is represented by D^N, D^{RD}, D^{RS} in Table 1. These quantities are only present as an additive term in the objective function and thus, to conform with the requirements of geometric programs, they are required to be posynomials.

For our implementations of the select and the symmetric hash join operators the additional delay introduced in generating the first output tuple (over the delay in obtaining the first input tuple from the upstream operators) either when processing normally or during recovery is negligible and so we equal them to zero. For the case of blocking aggregates, the additional delay introduced by the aggregate could be significantly large as discussed in Appendix B.4.1.

We now illustrate the approach by generating a model for Selects, Aggregates, and Joins. We first provide the missing details of Section 5.3.1 about the modeling of a symmetric hash join and then we present the detailed models for our remaining two operators: Select and Aggregate.

B.2 Join

We construct a simplified analytical model for our symmetric hash join. The model is: Given the cardinality of the input channels as $|I_1|$ and $|I_2|$ we assume that the input tuples belong to the first and the second relation with probability $p_1 = \frac{|I_1|}{|I_1|+|I_2|}$ and $p_2 = \frac{|I_2|}{|I_1|+|I_2|}$, respectively. We let $\hat{\sigma}$ represent the probability that a pair of tuples, one from each input relation, have the same join attribute. Let X_i be a random variable that denotes the number of output tuples produced due to processing the i^{th} input tuple. Thus, with this model, on receiving the i^{th} input tuple, the expected number of new output tuples produced is:

$$\begin{aligned}\mathbb{E}[X_i] &= \sum_{j=1}^{i-1} \mathbb{I}_{j,i} \\ &= \sum_{j=1}^{i-1} 2p_1p_2\hat{\sigma} \\ &= 2p_1p_2\hat{\sigma}(i-1)\end{aligned}$$

Here, $\mathbb{I}_{j,i}$ is an indicator random variable that is 1 when the tuple i joins with tuple j. Note that this happens when i and j belong to distinct input channels (with probability $2p_1p_2$, the multiplier 2 being for the two ways of exclusively assigning the tuples to the two input channels) and they have the same value for the join attribute.

Equating the expected number of total tuples generated after processing all the input to $\sigma|I_1||I_2|$ we get that:

$$\begin{aligned}\sum_{i=1}^{|I_1|+|I_2|} \mathbb{E}[X_i] &= \sigma|I_1||I_2| \\ p_1p_2\hat{\sigma}(|I_1|+|I_2|)(|I_1|+|I_2|-1) &= \sigma|I_1||I_2| \\ \implies \hat{\sigma} &= \sigma\frac{|I_1|+|I_2|}{|I_1|+|I_2|-1}\end{aligned}$$

Thus, after seeing n_{in} input tuples the expected number of total output tuples generated is:

$$\begin{aligned}NB_{out}(n_{in}) &= \sum_{i=1}^{n_{in}} \mathbb{E}[X_i] \\ &= \hat{\sigma}p_1p_2n_{in}(n_{in}-1) \\ &\approx \hat{\sigma}p_1p_2n_{in}^2\end{aligned}$$

We approximate the function $NB_{out}(n_{in})$ since for large values of n_{in} (which is very likely during large scale data processing) the approximation is very close to the actual function. Note that the approximation is required to conform to the requirements of a Geometric Program.

For joins, the other parameters of interest for different operating mode, *i.e.*, normal, during the recovery of the downstream, and during the recovery of the join) are discussed in Sections 5.3.2, 5.3.3, and 5.3.4, respectively.

B.3 Select

We model a select operator that has no output queues and can skip over input tuples during recovery.

B.3.1 Modeling Overhead of Fault Tolerance

A select operator with selectivity σ processes, on average, σ^{-1} input tuples to generate a single output tuple where each input tuple takes t^{cpu} time to process. For the strategy CHCKPT, since there is no output queue or state for select, each checkpoint is assumed to cost a fixed time (some multiple of t^{io} we refer to be t^{ckpt}.)

Thus,

$$t_a^{cpu} = \sigma^{-1}t^{cpu} + \mathbb{I}^M t^{io} + \mathbb{I}^C t^{ckpt}(c\sigma)^{-1}$$

where c is the number of tuples processed between consecutive checkpoints (a measure of checkpoint frequency.)

To understand how the values for γ and k are set, note that, on average, the select operator produces σn_{in} output tuples for every n_{in} input tuples it processes. Thus,

$$NB_{out}(n_{in}) = \sigma n_{in} = \gamma n_{in}^k$$

Hence, $\gamma = \sigma$ and $k = 1$.

We use a similar reasoning for deriving γ and k for other operators too.

Integration within the generic operator model: Once we have determined t_a^{cpu}, γ, and k we can plug these constants in the generic operator model template we derived in the previous subsection to get the model for the select operator.

B.3.2 Modeling Replay Request Times

For NONE and CHCKPT we process on an average σ^{-1} input tuples to generate each output tuple and each input tuple takes t^{cpu} time to process. For MATERIALIZE we read tuples from disk. Thus,

$$t_a^{cpu} = (\mathbb{I}^N + \mathbb{I}^C)t^{cpu}\sigma^{-1} + \mathbb{I}^M t^{io}$$

For strategies NONE and CHCKPT, we need to process the input tuples to generate the tuples to output for the downstream node. Thus the $NB_{out}(n_{in})$ function is identical to the case for normal processing. For MATERIALIZE, we read the output tuples from disk (the materialized tuples act as the "input" tuples for this recovery operation) and send them downstream and hence $NB_{out}(n_{in}) = n_{in}$.

Thus, the overall function is defined as follows.

$$\begin{aligned} \gamma &= (\mathbb{I}^N + \mathbb{I}^C)\sigma + \mathbb{I}^M 1 \\ k &= 1 \end{aligned}$$

B.3.3 *Modeling Recovery Time*

In the existing model we recreate everything from the upstream tuples. Thus the recovery output profile looks similar to the normal processing. Hence, $\gamma = \sigma$ and $k = 1$ and the minimum amount of work done per output tuple is: $t_a^{cpu} = \sigma^{-1} t^{cpu}$.

We will need to generate at most 1 tuple for NONE and MATERIALIZE and $c\sigma$ tuples for CHCKPT.

B.4 Aggregate

For the aggregate operators used in our experiments the final output fits in memory and that is the case that we model in this section. The aggregate operator as defined in this section can also perform aggregates after grouping the input tuples on a certain attribute. Note that we assume that the aggregate operator keeps the output tuples it has produced and sent downstream in memory (and by our previous assumption, this information can be stored in memory.)

The aggregate operator works in two distinct phases. In the first phase, no output is produced as the aggregated output tuples are incrementally computed, while in the second phase, the computed aggregates from the first phase are sent downstream. The second phase can be viewed as a select operator with selectivity one and that receives all of its input at an infinite rate and can process each input at a rate determined by the time it takes to access a tuple in memory. The time spent in the first phase is included as the delay terms D^N, D^{RD}, D^{RS}.

We take the checkpoint size to be $|I_u|\sigma$. Here, $|I_u|$ is the cardinality of the input tuples and σ is the selectivity of the aggregate. Note that there is an initial buildup phase, as the number of groups in the output are determined. For example, if there are 8192 groups, we will need to see at least 8192 input tuples before these groups are discovered. Since we experiment with output groups of sizes 8192 while the input tuples are in the order of a million tuples, we ignore this buildup phase and approximate the state to checkpoint to consist of 8192 tuples form the outset. The average state size for a blocking operator like sort will be half of the final output (since the size of the state will linearly increase with time.)

B.4.1 *First Tuple Delay*

To compute the delay in producing the first tuple, it should be noted that an aggregate first processes all of its input tuples before producing the first tuple. Further, in case of CHCKPT, the operator also takes checkpoints of the aggregated state before the first tuple is produced and thus the production of the first tuple is further delayed. As we assumed during the analysis of the Select operator, we assume that each checkpoint incurs an extra cost of t^{ckpt}.

$$D^N = |I_u| \max(x_u^N, t^{cpu}) + \mathbb{I}^C c^{-1} |I_u| (|I_u|\sigma) t^{io} + \mathbb{I}^C t^{ckpt} |I_u| c^{-1}$$

The first term represents the total amount of time required to only process the input tuples; the second term represents the amount of time required to only write out the total size of all checkpoints to disk, assuming sequential IO; and the last term is the seek time at the beginning of each checkpoint multiplied by the total number of checkpoints taken.

B.4.2 *Modeling Overhead of Fault Tolerance*

Before we model the overhead of fault tolerance we note that the aggregate operator is blocking and works in two stages: in the first the tuples are processed and aggregated while in the second the aggregated tuples are sent downstream. Since the operator is blocking, the time to produce the first tuple (captured by the term D^N) accounts for the first stage. For the second stage, we start with the set of aggregated tuples (grouped-by certain attributes) and send them downstream. It is this second stage that we model now. The output tuples are produced only after processing all the input tuples. Each output tuple takes t^{cpu} time to be processed. The processing required by each tuple to output is essentially the cost of looking it up in memory and sending it downstream. We take t^{cpu} to be this cost per tuple.

$$\begin{aligned} t_a^{cpu} &= t^{cpu} + \mathbb{I}^M t^{io} \\ \gamma &= 1 \\ k &= 1 \end{aligned}$$

The γ and k values are each 1 since each output tuple corresponds to an aggregate on exactly one grouped-by attribute (that was computed in the first stage of operation) and thus, $NB_{out}(n_{in}) = n_{in}$. Materialization of output incurs an extra cost of t^{io}.

B.4.3 *Modeling Replay Request Times*

When a downstream node requests a replay of older tuples, the aggregate resends the requested tuples from memory. Thus, the processing is equivalent to the processing of the second stage of the aggregate operator as observed in the normal mode of processing. Hence,

$$\begin{aligned} t_a^{cpu} &= t^{cpu} \\ \gamma &= 1 \\ k &= 1 \end{aligned}$$

B.4.4 *Modeling Recovery Time*

To model the recovery time for the aggregate operator it is necessary to know when the failure occurred since the recovery times, on average, differ in the two different stage of the computation. This is because if a failure occurs randomly in the first stage, the amount of work lost is half of the total (in expectation.) On the other hand, if the failure occurs in the second half, the loss is always the total work.

If the blocking operator is the last operator in the query plan (*i.e.*, top operator), or if the operator is selective (an example of a non-selective blocking operator is *sort*) then the amount of time spent in the second stage is negligible and we can assume that the operator fails only in the first stage. We derive our model using this assumption and discuss the modifications required for the cases where the assumptions are violated later.

In the case of CHCKPT, the operator reads tuples from disk and recreates the state. For this recovery operation the input are the tuples in the checkpointed state and the output is the state in memory. If t^{io} is the time it takes to read one tuple from disk then with CHCKPT,

$$t_a^{cpu} = t^{io}$$

and since each tuple read from disk becomes a part of the state in memory,

$$\begin{aligned} \gamma &= 1 \\ k &= 1 \end{aligned}$$

Note that the above discussion is only applicable for CHCKPT, for NONE and MATERIALIZE there is no state to read back. In the case of the CHCKPT the average state size has to be read back into memory which is of size $|I|\sigma$. After reading this state in memory the operator also needs to process the input tuples from the upstream nodes that were processed before but whose change on the in-memory state was not checkpointed. On an average, we need to process $\frac{1}{2}c$ tuples with a cost of t^{cpu} per tuple. We include this extra cost of $\frac{1}{2}ct^{cpu}$ to the delay term for recovery (D^{RS}).

Since we don't generate any output tuples, we represent the time taken to recover using the delay function D^{RS}. Upon failure, NONE and MATERIALIZE need to reprocess, on an average, half the input tuples, and CHCKPT needs to reprocess, on average, half of the tuples it processes between consecutive checkpoints. Thus,

$$D^{RS} = \left((\mathbb{I}^N + \mathbb{I}^M)\frac{1}{2}|I_u| + \mathbb{I}^C 0.5c \right) \max(x^{RD}, t^{cpu})$$

When the blocking operator is not at the end of the pipeline, the only modification required in our model derivation is to change the expected number of tuples to reprocess. As the time spent in the second stage increases, it becomes more likely that on failure NONE and MATERIALIZE have to reprocess their entire input to recover state and that the CHCKPT has to only read the checkpointed state to recover. To deal with this situation:

1. The optimizer can be used to get an lower and a upper bound on the expected runtime by assuming first that failures only occur during the first stage and then that the failures only occur during the second stage.
2. Another alternative would be to make an estimate of relative runtime of the two stages. For example, if the aggregate operator is at the beginning of a pipeline (rather than at the end), it is more likely that upon a failure, the aggregate was in its second stage; in such a situation we can take the expected tuples to be reprocessed to be the entire input (or the entire checkpoint, for the strategy CHCKPT.) Note that the time to send the tuples will not change. After sending the tuples, the operator will sit idle. But if it fails, recovering it is equivalent to re-evaluating all the tuples and regenerating the state.

Note that the presence of blocking operators does not affect the operator models for normal processing and when the downstream operator fails.