A Latency and Fault-Tolerance Optimizer for Parallel Data Processing Systems

Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska

University of Washington, Seattle, USA
{prasang,yongchul,magda}@cs.washington.edu

***Abstract*— An important problem faced by today's parallel data processing systems is fault-tolerance: how to handle failures during the execution of long-running queries. Existing systems either restart queries when failures occur or materialize intermediate results in a blocking fashion. The problem is that the former leads to expensive recovery while the latter slows-down normal processing and prevents pipelining.**

In this paper, we address the above problem of intra-query fault-tolerance in parallel data processing systems. Our key hypothesis is that a parallel system can achieve best performance by supporting efficient fault-tolerance strategies that do not require blocking but also by supporting the use of *different strategies at different operators within a single query plan*. Enabling each operator to use a different fault tolerance strategy leads to a space of possible fault tolerance plans amenable to cost-based optimization.

To test our hypothesis, we first develop a framework that enables the mixing and matching of fault-tolerance techniques in a single query plan. Second, we develop FTOpt, a cost-based fault-tolerance optimizer that, given a failure model, 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 different recovery techniques outperform any uniform strategy, and (3) our optimizer is able to correctly identify the winning fault-tolerance configurations.

I. Introduction

The ability to analyze large-scale datasets has become a critical requirement for modern business and science today. To carry out their analyses, users are increasingly turning toward parallel database management systems (DBMSs) [1], [2], [3] and other parallel data processing engines [4], [5], [6] deployed in shared-nothing clusters of commodity servers. In these clusters, the data is distributed and possibly replicated across tens to thousands of servers, and each analysis task executes in parallel across all or a subset of these servers.

In many systems, users can express their data processing needs using SQL or other specialized languages (*e.g.*, Pig Latin [7], DryadLINQ [8]). The resulting queries or scripts are then translated into a directed acyclic graph (DAG) of operators (*e.g.*, relational operators, maps, reduces, or other [6]) that execute in the cluster.

An important challenge faced by these large-scale data processing systems is fault-tolerance. When running a parallel query at large scale, some form of failure is likely to occur during query execution [9]. Existing systems take two radically different strategies to handle failures: parallel DBMSs adopt the transaction-based approach and 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 [4] and similar systems [5] 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 can significantly increase query runtimes 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 [10].

In this paper, we study the problem of making parallel and distributed query plans fault-tolerant in a manner that delivers high performance both under normal operations and when failures occur, and that introduces no new blocking points in the plan. Several specific objective functions are possible (*e.g.*, minimize total runtime with failures, minimize runtime without failures subject to a constraint on failure recovery time, etc). We choose to *minimize expected total runtime in the presence of failures*. That is, for a given query and expected operator failure rate, we seek to minimize the expected query runtime (*i.e.*, sum of time under normal processing and time spent in recovery). This function combines high-performance at runtime with fast failure recovery into a single objective.

We observe that data materialization is only one of several strategies for achieving fault-tolerance. Other strategies are possible including restarting a query or operator but skipping over previously processed data [11], [12] or checkpointing operator states and restarting from these checkpoints [13], [11]. We show that the fault-tolerance strategy which delivers the highest performance (*i.e.*, the lowest expected runtime with failures) depends on the query plan. Furthermore, we demonstrate that strategies which *mix-and-match different fault-tolerance techniques within a single query plan* can often outperform ones that apply a single technique across all operators. For example, an expensive aggregate operator may need to checkpoint its state while an inexpensive filter may best run without fault-tolerance and simply skip over previously processed data after a failure.

Given the above observations, we develop (1) a framework that enables *mixing-and-matching of fault-tolerance techniques in a single query plan* and (2) *FTOpt, a cost-based fault-tolerance optimizer* for this framework. Given a query plan and information about the cluster and expected failure rates, FTOpt automatically selects the best 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. We show how to integrate three fault-tolerance techniques from the literature (*i.e.*, materialization, checkpointing, and doing nothing) into our framework in a manner that does not introduce any new blocking points into a query plan, enabling users to retain the ability to see results incrementally. In summary, we make the following contributions:

1) *Extensible framework for heterogeneous fault-tolerance strategies*. 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 IV).
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 latency to complete the query given an expected number of failures (Section V).
3) *Operator models for pipelined plans*. Finally, 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 [14]. We also develop an approach that simplifies the modeling of other operators within our framework thus simplifying extensibility (Section V-B).

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), 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 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 is thus an important component of parallel data processing, enabling performance gains similar in magnitude to several other recently proposed MapReduce optimizations [15], [16].

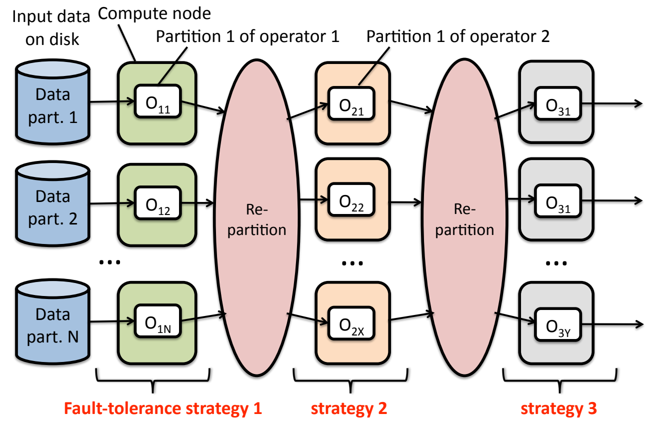

Fig. 1. Parallel query plan comprising three operators (O_1, O_2, O_3) and one input from disk. Each operator is partitioned across a possibly different number of nodes. Data can be re-partitioned between operators. Fault-tolerance strategies are selected at the granularity of operators.

II. Model and Assumptions

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, since we want to preserve pipelining when possible, we assume that a query plan takes the form of a tree (rather than a DAG) and that all operators are scheduled and executed at the same time. Importantly, however, even though we preserve pipelines, our optimizer works with both blocking and non-blocking operators.

In a shared-nothing cluster, different types of failures can occur. In this paper, to simplify the presentation and discussion, we focus only on *process failures*. That is, 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. However, multiple processes –and thus operators– can fail at the same time. The only requirement is that these operators be restarted from downstream to upstream.

Our approach could be extended beyond process failures with some changes. First, if entire physical machines can fail, then checkpoints must be written to remote nodes [17] incurring extra network and CPU costs that must be taken into account by the optimizer. Second, when a physical machine fails, the number of nodes in the cluster is reduced by one, which must also be taken into account. A network failure can cause one or more machines to become disconnected. It can also lead to network partitions. In both cases, our approach still works if a cluster management system ensures that only a fully connected set of machines continues processing the query. Larger-scale failures can also cause either input data or checkpointed data to become unavailable. In that case, the query would need to be restarted in its entirety once the input data became available again. In general, however, large-scale rack and network failures are typically 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) [18]. In this paper, we thus focus on process failures.

III. Background and Related Work

Fault-Tolerance in Relational DBMSs. Commercial relational DBMSs provide fault-tolerance through replication [19], [20], [21]. Similarly, parallel DBMSs [1], [2], [3] use replication to handle various types of failures. Neither, however, provides intra-query fault-tolerance [22].

Main-memory DBMSs [23], [24], [25] 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. Recently, a new generation of massively parallel data processing systems has been introduced [4], [5], [6], [7]. The MapReduce framework [4] 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 [6], 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 [26] 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 [10], 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 [13], [11], [27]. All fault-tolerance 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 [28], [29], [27]. 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 [13], [11], 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 IV-B).

Recently, Simitsis et. al. [30] 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.* [17] 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 [31], [32] use rollback recovery but are otherwise orthogonal to our work.

The Phoenix/App project [33] explores the problem of heterogeneous fault-tolerance in the context of web enterprise applications. For this, it defines an *interaction contract* between each pair of component types. In contrast, we define a single contract type that can hide the fault-tolerance method chosen by each operator.

IV. 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 [11], [27] and "contract-based" methods for query suspend-resume [31]. 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.

A. 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 [11], [27]. 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:

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 mask failures from client applications, except possibly for visible delays. They also enable operators to be individually restartable. These rules also enable a 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 can work within the pre-defined types of interactions.

From the above four rules, only the "Producer replay guarantee" rule potentially adds overhead to the system since it requires that a producer be able to re-generate (part of) its output. 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 restarted as well. This approach is equivalent to restarting a subset of the query plan after a failure occurs and is no worse than what parallel databases do today. Alternatively, an operator could write its output to disk, which adds overhead but speeds-up recovery of downstream operators. Finally, some operators, such as joins or aggregates, can easily re-generate their output from their state without the need for an output queue. 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.

B. Concrete Framework Instance

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

Our framework also requires that operators be deterministic. We develop a low-overhead method to achieve this goal, but we omit it here due to space constraints and refer the reader to our technical report for details [34].

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 [11].

As in upstream backup, operators such as selections or projections that do not maintain any state between consecutive tuples (*i.e.*, "stateless operators") can send acknowledgments in some cases. For example, if an input tuple r makes it through a selection operator to generate the output q and is acknowledged by all operators downstream, then r can be safely acknowledged. Unlike upstream backup, which uses different types of acknowledgments [11], our approach uses only one type of acknowledgments facilitating heterogeneous fault-tolerance. This approach of skipping over old input data during recovery has also been used for resumptions of interrupted warehouse loads [12].

To handle a request for output tuples, a stateless operator can simply fail and restart itself to reproduce the requested output. For expensive stateful operators (*i.e.*, operators such as joins and aggregates that maintain state between consecutive tuples), a more efficient strategy is to maintain an output queue and replay the requested data [11]. 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 directly from their state. We implement this strategy and use it in our evaluation.

Strategy CHCKPT. This strategy is a type of *rollback recovery* strategy where operators save their state periodically to stable storage. Because our framework recovers operators individually, it requires what is called *uncoordinated checkpointing with logging* [13]. One approach that can directly be applied is *passive standby* [11], where operators take periodic checkpoints of their state, independently of other operators.

Our framework requires that an operator saves sufficient information to guarantee the consumer progress, consumer durability, and producer replay guarantees. That is, the operator must log its state and, if it maintained an output queue, it must log that queue as well. After each such checkpoint, the operator can acknowledge the checkpointed input tuples. Upon failures, the operator restarts from its most recent checkpoint. As an optimization, operators can checkpoint only delta-changes of their state [13]. Other optimizations are also possible [13], [17], [35] and can be used with our framework.

Strategy MATERIALIZE. An alternate rollback recovery approach consists in logging intermediate results between operators as in MapReduce [4]. 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.

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

V. FTOpt

FTOpt is a fault-tolerance optimizer for our heterogeneous fault-tolerance framework: it selects the fault-tolerance strategy that should be used by each operator in the plan to minimize an objective function (*i.e.*, the expected runtime with failures) given a set of constraints (that model the plan). The challenge is to keep the optimizer tractable yet make it sufficiently accurate to select good fault-tolerance plans and make it easily extensible with new operators and fault-tolerance strategies. We present FTOpt's high-level structure and develop a detailed operator model.

A. Overview

FTOpt takes as input three pieces of information: (a) the tree-shaped query plan to optimize, (b) information about the cluster resources, and (c) models for the operators in this plan under different fault-tolerance strategies. FTOpt produces three outputs: (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. The latter is necessary because fault-tolerance strategies and resource allocation are intertwined. Due to space constraints, 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 each node. For the resource allocation details, we refer the reader to our technical report [34].

Objective Function. FTOpt minimizes the following cost function, that captures the expected runtime of a query:

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

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 time spent in recovery from failures. The cost function makes two assumptions. First, the entire pipeline (all paths) is blocked during a failure recovery since even if one operator partition fails, operators upstream from that partition stop their normal processing and participate in its recovery. As a result, they block everything else downstream and upstream. Second, all partitions of a failed operator are recovered after any partition fails. Since the pipeline blocks during recovery, the runtime is indeed similar whether one or all partitions need to be recovered.[1]

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 depth 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 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.

For the second term, $\mathcal{O}$ is the set of all operators in the tree. For operator $i \in \mathcal{O}$, z_i is its expected number of failures during query execution, and $\mathbf{R_i}$ is its expected recovery time from its failure. To compute z_is, we use an administrator-provided statistic nF: the expected number of process failures for the query. If n_i is the total number of processes allocated to operator i we assume that $z_i = nF \frac{n_i}{\sum_{j \in \mathcal{O}} n_j}$. nF can be estimated from the observed failure rates for previous queries and administrators typically know this number [9]. We assume nF to be independent of the chosen fault tolerance plan. nF 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 nF's value in Section VI-F).

Abstract Operator Model. To compute the objective function, FTOpt requires that each operator provides expressions that characterize its various delays and processing times 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 I expressed in terms of the parameters, represented by Θ in Table II. In Section V-B, we show how to, fairly easily, express such functions in our framework. Compared to existing cost models for parallel query runtime estimation [36], [37] and fault-tolerance in streaming engines [11], our models capture the dynamic operator interactions in pipelined queries, which we observed to affect query 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 [38], [39].

Constraints. Individual operator models are composed together to generate the model for the query by imposing three constraints during composition: (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 engine 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 V-B).

[1]Our implementation recovers only the failed partition while our optimizer assumes the entire operator fails. Experimental results shows that this approximation is sufficiently accurate for fault-tolerance optimization.

TABLE I

FUNCTIONS CAPTURING OPERATOR BEHAVIOR.

Delay to get 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 II

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.

Search Space. For a given query plan, the optimizer's search space consists of all possible combinations of fault-tolerance strategies. In this paper, we use a brute-force technique to enumerate through that search space. We leave more efficient enumeration algorithms for future work. For each fault-tolerance configuration, the optimizer computes optimal checkpoint frequencies and optionally an allocation of resources to operators as explained in our technical report [34].

B. Operator Modeling

We use a geometric programming(GP) framework to model the operators and optimize the fault-tolerance plan. We use GP since it allows expressions that model resource scaling and non-linear operator behavior but still finds a global minima for the model [14]. We first present our overall optimization problem and then derive our concrete operator models.

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

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

The optimization is then expressed as follows:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \le 1, \quad i = 1, \ldots, m \\ & g_i(x) = 1, \quad i = 1, \ldots, p \end{aligned}$$

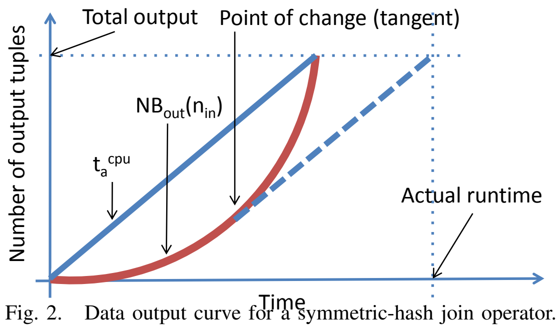

Fig. 2. Data output curve for a symmetric-hash join operator.

In our case, $x = \cup_{i=1}^{i=\mathcal{O}}\{x_i^N, x_i^{RD}, x_i^{RS}, c_i, N_i\}$ where $\mathcal{O}$ is the set of operators and the other variables come from Tables I and II. Our objective function is $f_0(x) = T_{total}$ as defined in Section V-A ("Objective function"). Finally, our constraints comprise framework constraints as defined in Section V-A ("Constraints") and operator constraints as we discuss next.

We now derive the detailed constraint equations for our join operator, which is the most complex of our three operators. The models for select and aggregate are similarly derived. They have been omitted due to space constraints, but can be found in the technical report [34].

2) Modeling Basic Operator Runtime: For our GP optimization problem, we must first derive the operator output rate (given by the inter-output-tuple delay, x^N) in the absence of failures and given an input rate for all its input streams.

We must also derive the delay value, D^N, but this delay is simply either negligible for selects and the symmetric hash-join that we model in our prototype or equal to the entire runtime for aggregates.

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 the more output it produces. 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 constraint for the next operator.

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 assuming that 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). 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. $NB_{out}(n_{in})$ 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. In this case, for a given selectivity σ, $\hat{\sigma} = \frac{|I_1||I_2|\sigma}{(|I_1|+|I_2|)(|I_1|+|I_2|-1)}$ is the probability that a new input tuple joins with an older tuple. In this case, the function $NB_{out}(n_{in})$ is defined as follows:

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

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. Informally, as the operator sees more input tuples, the number of the output tuples produced after processing a new input tuple should never decrease.

These inequalities take x^{IN} as input, which is the time interval at which input tuples are assumed to arrive. 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.

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

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

We refer the reader to the technical report for the exact derivation of this model [34]. 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 $NB_{out}(\cdot)$ 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 demonstrate the feasibility and impact of fault-tolerance optimization. We leave extensions to more complex models, including data skew between partitions, for future work.

3) Modeling Overhead of Fault-tolerance: We now extend the above models to take fault-tolerance overhead into account. For brevity, we use the notation that $\mathbb{I}^N$, $\mathbb{I}^M$ and $\mathbb{I}^C$ are either 0 or 1 depending on whether NONE, MATERIALIZE or CHCKPT is chosen as the fault tolerance option respectively.

The overhead of fault tolerance affects the minimum time an operator needs to produce an output. The exact overhead, and hence the model, depends on the implementation of the operator. An operator may checkpoint incrementally, or may checkpoint the entire state each time, may or may not checkpoint its output queue, etc.

For our implementation of joins, for MATERIALIZE we write to disk all the outputs produced and for CHCKPT, we only log the incoming tuples to disk incrementally. We do not maintain any output queue.

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.

4) Modeling Recovery Time for Downstream Neighbors: To estimate the replay time when a downstream node requests a range of tuples, we need to know the average rate at which the output tuples are produced to satisfy the request and the delay in generating the first requested tuple. Note that the replay rate might depend on when, during the course of the query, the downstream fails. For example, if the replay behaves in the same way as 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 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. A non-exhaustive list of choices for the operator is to maintain and use an output queue, use materialized output, use in-memory state, or reprocess from scratch without doing anything extra for fault tolerance.

For our join implementation, we use the in-memory hash table to regenerate the output. Hence the delay is going to be 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 use the same 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 arrival rate of inputs x^{IN} is at most t^{cpu} and we take $x^{IN} = t^{cpu}$.

5) 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 will depend on the implementation. A non-exhaustive list of choices for an operator is to recover from the last checkpoint, recover from the upstream nodes, recover partially by recovering the state but not an output queue, etc.

For our join implementation, 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 [34]), while CHCKPT rebuilds it from the input tuples logged to disk.

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

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.

C. Approach Implementability

In summary, 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 IV-B, however, we showed, how to efficiently implement three well-known fault-tolerance strategies for generic stateless and stateful operators. There also exist libraries that can help with such implementation (*e.g.*, [35]). 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 [2], [1] and MapReduce-type systems such as Hive [40] or Pig [7], which all come with a pre-defined set of 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 VI-D.

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.*, [41]), 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 V-A), operator selectivities (standard optimizer-provided metric), and an estimate of the total checkpointable state. We show in Section VI-F that our optimizer is not sensitive 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.

VI. 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.[2] 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. In all experiments, a subset of cores and disks are dedicated to each operator partition. The cluster is running a very 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 as described in Section IV-B. We implemented the optimizer in MATLAB using the *cvx* package [42]. The queries that we use (Table VI-B) are standard relational queries and process synthetically generated data without skew. Tuples are 0.5 KB in size. 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.

A. 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 $t^{io} = 13.0\ \mu s$ 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 us now to predict runtime for an entire query plan. Figure 3 shows the runtime without failure for a few two-operator queries. While the median percentage difference between real

[2]17 was the largest number of machines that we could reserve for our experiments in our local database cluster.

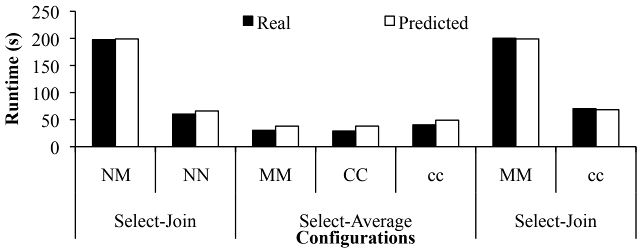

Fig. 3. Runtime without failures for different two-operator queries. The x-axis labels indicate the fault tolerance strategy chosen: N for NONE, M for MATERIALIZE, C for CHCKPT with a total of 10 checkpoints and c for CHCKPT with 1000 checkpoints.

TABLE III

Query parameters in terms of tuples

The entries are the number of tuples				
Query 1	Op1: **Select**	Op2: **Join**	Op3: **Join**	Op4: **Join**
Input	2×10^6	4×10^6	4×10^6	16×10^6
Output	2×10^6	2×10^6	8×10^6	8×10^6
Query 2	Op1: **Select**	Op2: **Join**	Op3: **Join**	Op4: **Join**
Input	1.6×10^8	16×10^6	16×10^6	16×10^6
Output	8×10^6	8×10^6	8×10^6	8×10^6
Query 3	Op1: **Select**	Op2: **Join**	Op3: **Join**	Op4: **Aggregate**
Input	1.6×10^8	16×10^6	32×10^6	1.6×10^8
Output	8×10^6	16×10^6	1.6×10^8	8192
Query 4	Op1: **Select**	Op2: **Join**	Op3: **Join**	Op4: **Aggregate**
Input	1.6×10^8	16×10^6	80×10^6	80×10^6
Output	8×10^6	40×10^6	80×10^6	8192
Query 5	Op1: **Select**	Op2: **Join**	Op3: **Join**	Op4: **Join**
Input	1.6×10^8	16×10^6	16×10^6	16×10^6
Output	8×10^6	8×10^6	8×10^6	3.2×10^7
	Op5: **Select**	Op6: **Join**	Op7: **Join**	Op8: **Aggregate**
Input	3.2×10^7	17.92×10^6	4.02×10^7	7.94×10^7
Output	8.96×10^6	2.01×10^7	7.94×10^7	8192

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 VI-F.

B. 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 4 through 6 show the actual and predicted runtimes for Queries 1 through 3 from Table VI-B 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 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

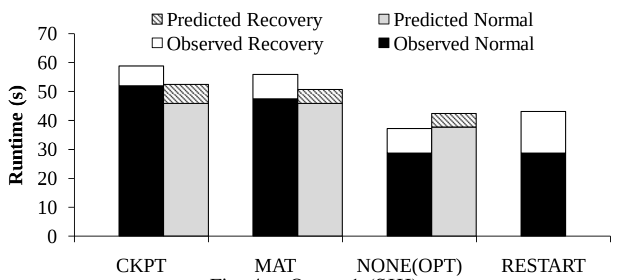

Fig. 4. Query 1 (SJJJ)

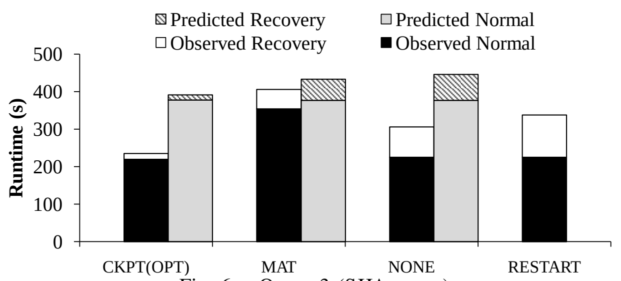

Fig. 6. Query 3 (SJJA query)

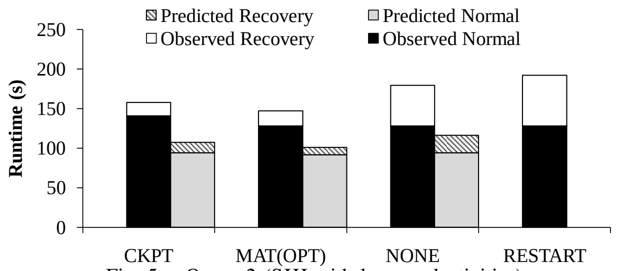

Fig. 5. Query 2 (SJJJ with lower selectivities)

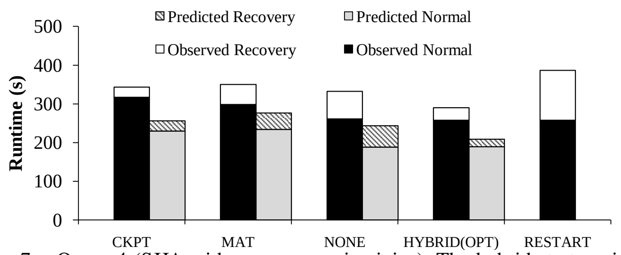

Fig. 7. Query 4 (SJJA with more expensive joins). The hybrid strategy is to materialize after select, do nothing for joins, and checkpoint the aggregate

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!* The predicted runtimes do not exactly match the observed ones. 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. 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 3 through 7, FTOpt preserves that order most of the time. In configurations where two fault-tolerance plans lead to very similar runtimes, it may not find the best plan due to its approximations but it 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.

C. Benefits of Hybrid Configurations

We now consider a query (Query 4), similar to Query 3, but with the joins processing and producing much more data, making checkpointing expensive. Figure 7 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 8 shows that the hybrid plan (chosen by the optimizer) materializes both the select's output, does nothing for the joins and takes 20 checkpoints of the aggregate. The best and worst uniform strategies and RESTART are 16%, 23% and 36% slower, respectively. We found, manually, that checkpointing the first two joins in the hybrid plan led to another hybrid plan that was 2% faster. Even though the optimizer doesn't choose the best plan, the plan it chooses performs similarly to the optimal one. Further, both the observed and the 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.

D. Performance in Presence of UDOs

We now 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 now treat its last operator, the aggregate, as a UDO that is not fault-tolerant and can only restart from scratch if it fails. Note that Rule 4.2 and operator determinism [34] allow restarting a UDO in isolation without restarting the entire query. The other operators remain unchanged and FTOpt is restricted to using NONE as the sole strategy for the UDO.

Figure 9 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, we find that materializing the first operator's output and using NONE for

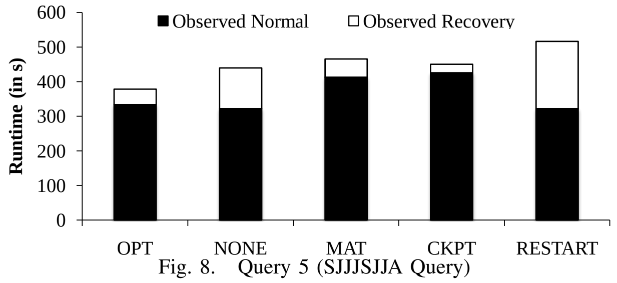

Fig. 8. Query 5 (SJJJSJJA Query)

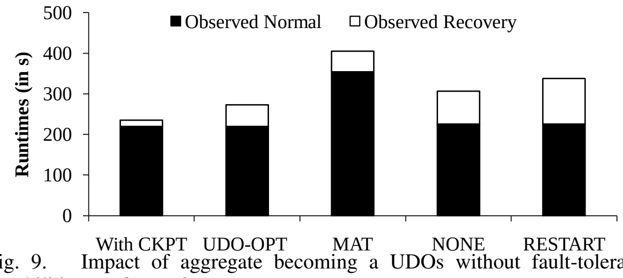

Fig. 9. Impact of aggregate becoming a UDOs without fault-tolerance capabilities on Query 3

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.

E. Scalability

FTOpt's MATLAB implementation uses the cvx package, which offers a successive approximation solver using SDPT3 [43]. In our current prototype, the average time to solve the optimization problem per plan is around 25s for the 4 operator plans in the previous sections. These slow results, however, are not an intrinsic limitation of our approach but a limitation of our prototype implementation. Indeed, the behaviour of an operator for a fault tolerance strategy is described using at most 12 inequality and 4 equality constraints and 11 variables. Modeling the network requires an extra inequality constraint per operator and one global inequality constraint. 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. Optimized solvers already exist that can solve such problems in under 1 ms [44].

We use a brute force search algorithm to find the optimal fault-tolerance plan. 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 the heuristic, the optimizer examines only 28 out of 81 configurations for Query 4.

Finally, the search algorithm essentially computes the least costly of a set of independent optimization problems and all of these problems can be optimized in parallel.

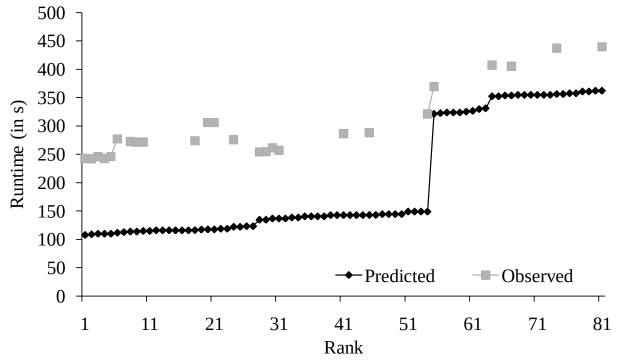

Fig. 10. The figure depicts the observed and predicted runtimes for Query 3, sorted on the predicted runtime, for all 81 fault tolerance plans for the query.

TABLE IV
REAL RANKINGS OF TOP 5 PLANS FROM PERTURBED CONFIGURATIONS.

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

F. Optimizer Sensitivity

We now evaluate FTOpt's sensitivity to inaccuracies in its parameters' estimates. We experiment with Query 3 since it is most sensitive to wrong choices: Figure 10 shows that runtimes vary from about 250s to 400s depending on the chosen plan.

To evaluate the sensitivity for a given parameter, we re-run 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 10). Table IV 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 IV 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 IO 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 ends up being more sensitive to perturbations in the selectivity estimates.

VII. 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 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, Dan Suciu, Bill Howe, and Julie Letchner for helpful comments on early drafts of this paper.

References

[1] "Greenplum database," http://www.greenplum.com/.

[2] "Teradata," http://www.teradata.com/.

[3] "Vertica, inc." http://www.vertica.com/.

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

[5] "Hadoop," http://hadoop.apache.org/.

[6] 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.*, 2007, pp. 59–72.

[7] 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.*, 2008, pp. 1099–1110.

[8] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. of the 8th OSDI Symp.*, 2008.

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

[10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. of the 7th NSDI Symp.*, 2010.

[11] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. of the 21st ICDE Conf.*, Apr. 2005.

[12] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik, "Efficient resumption of interrupted warehouse loads," *SIGMOD Record*, vol. 29, no. 2, pp. 46–57, 2000.

[13] 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*, vol. 34, no. 3, pp. 375–408, 2002.

[14] S. P. Boyd, S. J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," Stanford University, Info. Systems Laboratory, Dept. Elect. Eng., Tech. Rep., 2004.

[15] 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)*, 2010, pp. 181–192.

[16] 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, pp. 51–62.

[17] 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.

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

[19] 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.

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

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

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

[23] 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.*, 2006, p. 99.

[24] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," in *Proc. of the 5th ICDE Conf.*, 1989, pp. 452–462.

[25] M. V. Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White, "An evaluation of checkpoint recovery for massively multiplayer online games," in *Proc. of the 35th VLDB Conf.*, 2009.

[26] 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.

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

[28] 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.

[29] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

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

[31] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang, "Query suspend and resume," in *Proc. of the SIGMOD Conf.*, 2007, pp. 557–568.

[32] 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, pp. 735–745.

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

[34] P. Upadhyaya, Y. Kwon, and M. Balazinska, "A latency and fault-tolerance optimizer for parallel data processing systems," Univ. of Washington, Tech. Rep., 2010.

[35] 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.

[36] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *Proc. of the SIGMOD Conf.*, 1992, pp. 9–18.

[37] 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, pp. 172–181.

[38] W. Hasan and R. Motwani, "Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism," in *Proc. of the 20th VLDB Conf.*, 1994, pp. 36–47.

[39] M. Zaït, D. Florescu, and P. Valduriez, "Benchmarking the DBS3 parallel query optimizer," *IEEE Parallel Distrib. Technol.*, vol. 4, no. 2, pp. 26–40, 1996.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *Proc. of the 26th ICDE Conf.*, 2010.

[41] "Oracle 10g," http://www.oracle.com/technology/products/database/oracle10g/index.html.

[42] "Cvx," http://www.stanford.edu/~boyd/cvx/.

[43] R. H. Tütüncü, K. C. Toh, and M. J. Todd, "Solving semidefinite-quadratic-linear programs using SDPT3," *Mathematical programming*, vol. 95, no. 2, pp. 189–217, 2003.

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

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

II. Appendix

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

A. Ensuring Operator Determinism

Our framework requires that operators and their partitions be deterministic. In particular, rule 4.2 requires that, in response to a valid request, an operator (or operator partition) always returns the same sequence of tuples, irrespective of any failures of that operator.

Most relational operators can easily 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 ensuring a deterministic input-data order is to buffer and interleave tuples using a pre-defined rule [28], [45]. These techniques, however, can impose a significant memory overhead due to tuple buffering.

Instead, we adopt the approach of logging determinants [13]. As the operator receives input tuples, it accumulates them into small batches, with one batch per input relation partition. For example, an operator with two inputs could receive a total of 3500 tuples, starting with tuple id p_1^1, from parent 1 in one batch and a total of 4000 tuples, starting from p_1^2, from parent 2 in another, larger batch. The actual arrival of tuples might be interleaved. We buffer the 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, the logged entries might look as follows: $\langle 2, p_1^2, 2500\rangle, \langle 1, p_1^1, 2500\rangle, \langle 1, p_{2501}^1, 1000\rangle, \langle 2, p_{2501}^2, 1500\rangle$.

Log entries are force-written to stable storage but, as we show below, this logging overhead is negligible even for small 512-tuple batches. 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 an 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).

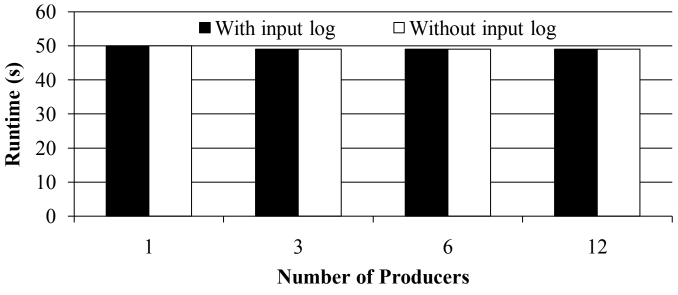

Fig. 11. Each pair of bars represents the time to complete processing, with and without logs, with a different number of upstream producers for a select operator. There is virtually no overhead even for 12 input streams.

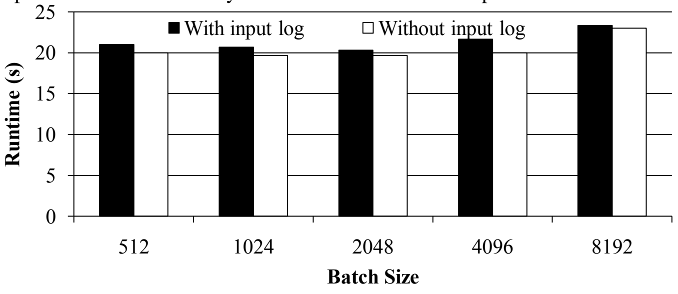

Fig. 12. Each pair of bars represents the time to complete processing, with and without logs, with different batch sizes for a join operator. The minimum overhead occurs with a batch size of 2048.

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.

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 11 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 12 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.

B. 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 also assumes 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 operator i takes t^{cpu} time to process a tuple, then with n_i machines it takes $\frac{t^{cpu}}{n_i}$ 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.

III. Operator Modeling Strategy

In this section, we present an approach for modeling any operator. We now illustrate the approach by generating a model for join. Subsequently we present the detailed models for our remaining two operators: select and aggregate.

The total time taken for a single operator to complete processing all its input tuples, when it receives input tuples at fixed and regular time intervals is determined by one of the following cases.

- **Case 1:** If the input tuples arrive slower than the rate at which the operator can process each one of them, then only the arrival rate determines the total completion time.
- **Case 2:** If the time taken to process each tuple is more than the time it takes a new input tuple to arrive then the time taken per tuple determines the total time to complete. The time taken to process each tuple may vary with time.
- **Case 3:** As seen in Figure 2 an operator may start in case 1 and then shift to case 2.

We now show how our optimizer uses a set of inequalities that characterize the average time interval between successive output tuples produced by an operator to compute the total time to completion for all three cases. 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. Informally, as the operator sees more input tuples, the number of the output tuples produced after processing a new input tuple should never decrease.

These inequalities take x^{IN} as input, which is the time interval at which input tuples are assumed to arrive. 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.

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

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

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. Specically, the rate at which output tuples are produced by the operator after time t_f is

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

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. The inequality becomes an equality when the operator operates in either of case 2 or case 3.

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 (case 1). 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 $NB_{out}(\cdot)$ 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. Specifically,

$$x^{IN} \times (NB_{out})^{-1}(S) + m_e^{-1} \times (|I| - S) \leq x^N |I|$$

where

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

Simplifying this we get,

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

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.

A. *First Tuple Delay*

The delay to produce the first tuple is represented by D^N, D^{RD}, D^{RS} in Table I. 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 do not discuss these cases. For the case of blocking aggregates, the additional delay introduced by the aggregate could be significantly large as discussed in Section III-C.1.

B. *Select*

We experimented with a select operator that had no output queues and could skip over input tuples during a recovery.

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,

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

Using the fact that, $NB_{out}(x) = \sigma x$ we get,

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

2) Modeling Recovery Time for Downstream Neighbor: 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,

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

Using the fact that we need to process inputs again for NONE and read tuples from disk for MATERIALIZE.

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

3) Modeling Recovery Time: The minimum amount of work done per output tuple. $t_a^{cpu} = \sigma^{-1} t^{cpu}$ In the existing model we recreate everything from the upstream tuples.

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

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

C. *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 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 assume that aggregates occur only at the end of a stage. The average checkpoint size is $|I|\sigma - 0.5|I|\sigma^2$.

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 for the case of select we assume that each checkpoint incurs extra cost of t^{ckpt}.

$$\begin{aligned} D^N = & |I| \max(x^i, t^{cpu}) + \\ & \mathbb{I}^C c^{-1} |I| (|I|\sigma - 0.5|I|\sigma^2) t^{io} + \\ & \mathbb{I}^C t^{ckpt} |I| c^{-1} \end{aligned}$$

2) Modeling Overhead of Fault Tolerance: The output tuples are produced only after processing all the input tuples. Each output tuple takes t^{cpu} time to be processed.

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

3) Modeling Recovery Time for Downstream Neighbors: This sections is irrelevant. The operator is always found at the end of a stage.

4) Modeling Recovery Time: For both NONE and CHCKPT, we need to reprocess some input tuples. We think of the state as the number of input tuples processed. Work done in processing one tuple is,

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

The input tuples arrive at intervals of x^{r_i} time units. Thus,

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

For CHCKPT, the number of tuples to reprocess, in expectation, is $((|I|\sigma - 0.5|I|\sigma^2) + \frac{1}{2}c$. The first term is expected size of the state to read into memory while the second is the expected number of tuples reprocessed. For the strategy NONE the expected number of input tuples reprocessed is $\frac{1}{2}|I|$.