Data Science and Engineering (2020) 5:65–79
https://doi.org/10.1007/s41019-019-00114-z

Smart Intra-query Fault Tolerance for Massive Parallel Processing Databases

Yunhong Ji[1] · **Yunpeng Chai**[1] · **Xuan Zhou**[2] · **Lipeng Ren**[1] · **Yajie Qin**[1]

Received: 11 June 2019 / Revised: 6 December 2019 / Accepted: 11 December 2019 / Published online: 19 December 2019
© The Author(s) 2019

Abstract
Intra-query fault tolerance has increasingly been a concern for online analytical processing, as more and more enterprises migrate data analytical systems from mainframes to commodity computers. Most massive parallel processing (MPP) databases do not support intra-query fault tolerance. They may suffer from prolonged query latency when running on unreliable commodity clusters. While SQL-on-Hadoop systems can utilize the fault tolerance support of low-level frameworks, such as MapReduce and Spark, their cost-effectiveness is not always acceptable. In this paper, we propose a smart intra-query fault tolerance (SIFT) mechanism for MPP databases. SIFT achieves fault tolerance by performing checkpointing, i.e., materializing intermediate results of selected operators. Different from existing approaches, SIFT aims at promoting query success rate within a given time. To achieve its goal, it needs to: (1) minimize query rerunning time after encountering failures and (2) introduce as less checkpointing overhead as possible. To evaluate SIFT in real-world MPP database systems, we implemented it in Greenplum. The experimental results indicate that it can improve success rate of query processing effectively, especially when working with unreliable hardware.

Keywords Intra-query fault tolerance · Fault tolerance · Pipeline · Massive parallel processing databases

1 Introduction

Massive parallel processing (MPP) databases are popular data platforms for enterprise and scientific data analysis. Compared with other types of platforms, such as Hadoop and Flink, the advantages of MPP databases lie in their maturity and comprehensive functionalities established in the last decades. Well-known MPP database products include Teradata [1], Greenplum [2], Vertica [3], as well as a number of SQL-on-Hadoop systems such as Impala [4] and HAWQ [5].

✉ Yunhong Ji
jiyunhong@ruc.edu.cn

Yunpeng Chai
ypchai@ruc.edu.cn

Xuan Zhou
zhou.xuan@outlook.com

Lipeng Ren
lipeng_ren@163.com

Yajie Qin
qinyajie@msn.com

[1] Renmin University of China, Beijing, China

[2] East China Normal University, Shanghai, China

In recent years, fault tolerance of query processing has become increasingly important to MPP databases. Instead of relying on expensive mainframes equipped with highly available hardware components, more and more enterprises are inclined to deploy their databases on cheap commodity machines. Commodity clusters are much less reliable than mainframes, such that databases have to deal with system failures proactively. Moreover, execution time of today's OLAP queries has expanded a lot, due to the rapid growth of data volume. The longer a query runs, the more it suffers from system failures [6].

To handle failures during the execution of a query, existing solutions typically adopt one of the following two strategies. The first strategy is to abandon the current round of execution and redo the entire query. Most existing MPP database systems adopt this strategy partly because they are not originally designed for platforms with high failure rate. If the query has to be completely rerun, it may lead to a severe delay. There is even a risk that a query will never finish, if system failures occur repeatedly.

The other strategy, which is adopted by some SQL-on-Hadoop or SQL-on-Spark systems, is to build the query processor on top of a fine-grained fault tolerance mechanism, such

as MapReduce [7] and Spark [8]. These mechanisms materialize the intermediate results of each query execution step, so that only the lost steps need to be redone when a failure occurs. However, this blind materialization approach is expensive. Its overhead is sometimes unnecessarily high, resulting in unacceptable query response time [9]. In fact, the benefit of materialization does not always pay off its cost. Moreover, it is difficult to apply this strategy to established MPP database systems, as it requires a complete reconstruction of the system.

To achieve intra-query fault tolerance, we have to materialize intermediate data of query processing somehow. However, a cost-effective approach will choose each materialization point carefully, lest it incurs too much overhead. It should also minimize the intervention to query processing pipelines, on which MPP databases rely to achieve good performance. Some existing approaches to intra-query fault tolerance [6, 10] do perform materialization selectively, aiming to maximize its cost-effectiveness. However, they either choose to block query processing pipelines [6] or hold unrealistic assumptions [10] that data transmission among query processing operators be order preserving. They do not appear practical for existing MPP databases. Besides, all these approaches emphasize performance over success rate—the probability of successfully finishing a query in a given time. In real world, enterprises may regard success rate more important, as they need to get work done in time, e.g., get the report in time .

In this paper, we propose a smart intra-query fault tolerance (SIFT) mechanism for MPP databases. SIFT always selects appropriate operators for intermediate result materialization, by considering all the factors mentioned above. It adopts a different optimization goal—maximizing query success rate. In summary, SIFT can be characterized from the following perspectives:

1. To minimize the negative impact on the performance of query processing, SIFT chooses not to block pipelines. To this end, it selectively materializes the blocking operators (or the endpoints) of pipelines only.
2. SIFT autonomously determines which query operator to materialize, by considering both the cost and the risk of system failure. We devised a novel optimization algorithm for selecting materialization points that allow the system to achieve a desired success rate by performing as little materialization as possible.
3. SIFT is light weighted and can be implemented in most of the MPP databases at an affordable engineering cost. As a proof, we implemented it in Greenplum, a widely used open-source MPP database.

In the experimental study, we show how SIFT enables Greenplum to achieve a certain degree of intra-query fault tolerance while preserving its performance in query processing. As our experiments suggest, with a 10% increase of query processing time, SIFT can help Greenplum cutoff 60% of the latency for handling a failure.

The rest of the paper is organized as follows. In Sect. 2, we review the conventional parallel query processing architecture and model. In Sect. 3, we introduce SIFT's basic approach to intra-query fault tolerance. The implementation of our approach in GP is detailed in Sect. 4. Section 5 presents results of our experimental evaluation. The related work is discussed in Sect. 6. Finally, we conclude the paper in Sect. 7.

2 Background

This section provides an overview about the architecture of MPP database (Sect. 2.1) and parallel query processing (Sect. 2.2). As the goal of SIFT is to maximize query success rate, a model about success rate is introduced in Sect. 2.3.

2.1 MPP Database Architecture

Typical MPP databases usually adopt a shared-nothing architecture [2], composed of one master node and n slave nodes. The master node is responsible for interacting with clients, managing the whole cluster and coordinating the query processing. Each of the n salve nodes is responsible for storing a partition of the data and performing query processing on its partition. Each slave node hosts d database instances, which will be referred to as *segments* subsequently. Figure 1 illustrates an example of this architecture, in which $n = 4$ and $d = 2$.

Most MPP databases provide fault tolerance at storage level. A mirror scheme, i.e., replication, is commonly used to ensure durability and availability of data. As Fig. 1 shows, each segment (*primary* segment) is allocated with a mirror (*mirror* segment) in another node. The master node detects node failures by monitoring heartbeats of slave nodes. If a slave node stops responding for a certain

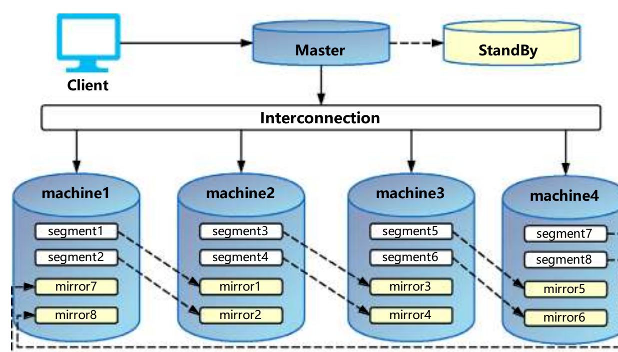

Fig. 1 Architecture of MPP databases. There are a master and 4 slaves, with 2 *segments* on each slave and a mirror for each *segment* on another machine.

amount of time, known as a system delay time (normally around 1 min), the master will treat it as a failed node. Once a failure is detected, the corresponding mirror will be activated to replace the failed primary segment. Similarly, a *standby* works as the replication/mirror of the master node. Through the mirror scheme, the system's availability can be significantly enhanced.

However, such a mirror scheme does not support intra-query fault tolerance automatically. When a node failure occurs, the running query's state on the failed node will be lost. After the corresponding mirror is activated, the whole query has to be rerun. If it is a long running query, response to the client will be severely delayed. In the worst case, a query will run indefinitely, if the probability of failure is high.

2.2 Parallel Query Processing

Typical MPP databases model a query plan as a directed acyclic graph [6, 10]. Each vertex in this figure corresponds to an operator, which is regarded as the unit of query processing. Typical operators include *scan*, *sort*, *aggregation*, *join*, etc. A query plan is produced by the master and broadcasted to all segments, which will execute the same plan on their own data.

Operators are usually executed in pipelines in MPP databases [10]. Pipelining can effectively reduce data materialization during query processing. Intermediate results will be immediately passed to the succeeding operator to process as soon as they are generated by the preceding operator.

In an MPP database, data need to be exchanged among slave nodes during query processing. Communication points break a query plan into multiple stages. On each segment, a query plan will be carried out by s concurrent processes or threads (further referred to as *slices*), each responsible for a stage of the query plan.

Figure 2 shows the actual query plan for the 17th query ($Q17$) of TPC-H benchmark [11] generated by Greenplum. The query is defined in Fig. 3. It is a typical query plan on a *segment*, with 15 operators and 4 *slices*. In this example, 4 processes or threads per segment will run in parallel to execute the query. Figure 4 shows the situation where Greenplum executes the query using 6 pipelines. It shows the processing time of each operator. For instance, in $P1$, intermediate results produced by $Op.1$ will be passed to $Op.4$ as soon as they are generated. Thus, we could see that the two operators overlap in executing time.

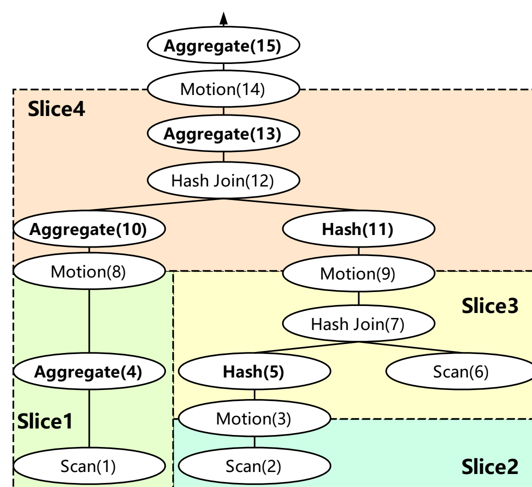

Fig. 2 Query plan for TPC-H $Q17$ generated by Greenplum

```
select
        sum(l_extendedprice) / 7.0  as  avg_yearly
from
        lineitem,
        part,
        (SELECT
                l_partkey AS agg_partkey,
                0.2 * avg(l_quantity) AS avg_quantity
         FROM lineitem
        GROUP BY l_partkey) part_agg
where
        p_partkey = l_partkey
        and agg_partkey = l_partkey
        and p_brand = 'Brand#31'
        and p_container = 'LG DRUM'
        and l_quantity < avg_quantity
limit  1;
```

Fig. 3 The 17th query($Q17$) of the TPC-H benchmark

2.3 Query Success Rate

For an OLAP query, users usually expect a latency constraint, to ensure the timeliness of decision making. Once a node failure occurs, execution of an ongoing query will be delayed. Our goal of intra-query fault tolerance is to make sure that a query can finish within its time constraint, even when confronted with failures. We consider success rate as the probability of successfully finishing a query within a time constraint T. In the following, we show how to calculate success rate.

2.3.1 One-Time Success Rate

One-time success rate refers to the probability of successfully finishing a query in a single round of execution, i.e., no failure happens during the execution.

Let the mean time between failures (MTBF) of a single node be $MTBF$. Then, the MTBF of an entire cluster

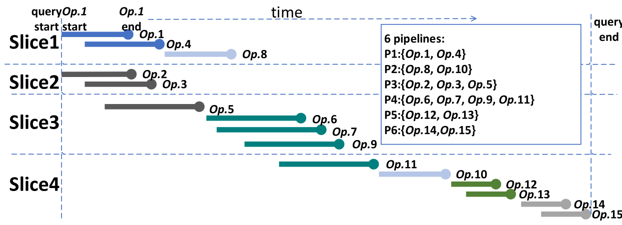

Fig. 4 Situation of executing in pipelines for $Q17$. There are 6 pipelines

containing n nodes is $\frac{MTBF}{n}$. We assume that node failures are independent. Then, intervals between node failures are subject to exponential distribution. In other words, the probability that a failure occurs within a time interval t can be calculated by $1 - e^{-\lambda t}$, where λ is $\frac{1}{MTBF}$. As what most database systems do, we estimate the cost of query processing by the number of I/O operations. Thus, the λ of an MPP database can be calculated as:

$$\begin{aligned} MTBF_{IO} &= \frac{S_{IO}}{\theta \times MTBF} \\ \lambda &= \frac{n}{MTBF_{IO}} \end{aligned} \tag{1}$$

In the equation above, θ is the throughput of disk and S_{IO} is the block size of each I/O operation. $MTBF_{IO}$ is the average number of I/O operations that occur between two consecutive failures. The parameter λ can then be deduced from $MTBF_{IO}$.

Let t_0 be the total time an MPP database takes to execute a query without encountering a failure. Then, the one-time success rate can be calculated as: $e^{-\lambda t_0}$.

2.3.2 Success Rate Without Fault Tolerance

For a system without intra-query fault tolerance, success rate with given time T, $Succ(T)$ can be calculated as follows:

$$\begin{aligned} & Succ_0(T) = e^{-\lambda t_0} \\ & Succ_i(T) = f_i(t_0, T) * e^{-\lambda t_i}, \quad i \in \{1, 2, \dots\} \\ & t_i = t_0, \quad i \in \{1, 2, \dots\} \\ & f_i(t_0, T) = \int \dots \int_D \prod_{j=1}^{i} \lambda e^{-\lambda x_j} dx_1 \dots dx_i, \\ & D = \{(x_1, \dots, x_i) | \sum_{j=1}^{i} x_j \le T - t_0\} \\ & Succ(T) = Succ_0(T) + Succ_1(T) + Succ_2(T) \\ & \qquad + \dots + Succ_i(T) + \dots \end{aligned} \tag{2}$$

Table 1 Parameters and description

Param	Description
$MTBF$	Mean time between failures
$MTTR$	System delay used for the system to recover after a failure
n	Number of physical machines
λ	Parameter of the failure model of the cluster
t_0	Total time a query used to run once without fault tolerance
t_{SIFT}	Total time a query used to run once with fault tolerance
Cr_i	Running cost of operator i without fault tolerance
Cm_i	Cost of operator i need to be materialized for fault tolerance
Mf_i	Materialization factor of operator i
SR_T	Target success rate for given time T

In the equations above, $Succ_i(T)$ represents the success rate after encountering i failures. t_i denotes the time needed to finish the query after i failures. As the system does not support intra-query fault tolerance, we have to redo the query from the beginning whenever there is a failure. Thus, the time we need to finish a query is still t_0 after i failures. The success rate after i failures will be $e^{-\lambda t_0}$. In Eq. 2, $f_i(t_0, T)$ is the probability of i failures occurring within time of $T - t_0$. As we can see, the success rate decreases as t_i increases. Intra-query fault tolerance enables us shorten t_i and thus improve the success rate.

Table 1 shows the parameters we use in this paper.

3 The SIFT Mechanism

When a failure occurs, we expect that the query can be resumed from an intermediate state rather than being re-executed from the very beginning. This requires us to perform checkpointing during query execution, i.e., selectively saving intermediate states into the persistent storage of mirrors.

As mentioned earlier, it is not cost-effective to perform checkpointing arbitrarily. Instead, two steps are required to make an appropriate checkpointing plan. First, a set of query operators are selected as checkpoint candidates (see Sect. 3.1). Second, success rates of possible checkpointing plans are evaluated and an optimal one is finally chosen (see Sect. 3.2).

3.1 Pipeline Preserving Checkpointing

Injection of checkpoints into a query plan may have two-sided effects. Firstly, it may break the pipeline. Secondly, extra cost will be introduced for materialization. To minimize performance penalty, SIFT's checkpoint candidates must satisfy the following conditions.

1. They should be breaking operators, i.e., those at the end of pipelines. This does not break the original pipelines. Meanwhile, as the intermediate results of these operators will anyway be materialized locally, the overhead can be minimized.
2. They must be deterministic operators, i.e., their intermediate results are uniquely determined by the data and the query plan. This guarantees the correctness of fault tolerance. In other words, it guarantees that we will reproduce correct results from checkpoints.

For instance, query operators in Greenplum include *scan*, *limit*, *sort*, *aggregation*, *motion*, *hashjoin*, *nestloopjoin*, *sortjoin* and etc. Among them, *motion* is a special type of operator responsible for exchanging data. According to the above rules, three kinds of operators can be candidates of checkpoints. They are the *hash* operator within *hashjoin*, *sort*, and *aggregation*. They are all endpoints of pipelines. Besides, they are all common operators in query plans.

Upon a failure, the master will replace failed segments with their mirrors and restart queries from their latest checkpoints. Figure 5 explains the effects of intra-query fault tolerance. It uses the same query as Fig. 3. As shown in Fig. 3, the query plan consists of 4 slices and 6 pipelines. In the plan, Operators 4, 5, 10, 11, 13, and 15 are checkpoint candidates. We assume that all 6 checkpoints are activated. Figure 5 shows a situation where a failure occurs after finishing checkpointing *Op*.4 and *Op*.5. During rerunning, Operators 1–5 can be skipped. The query execution can start from intermediate results of *Op*.4 and *Op*.5. As a result, *Slice* 2 does not need to be launched at all. Meanwhile, the work of *Slice* 1 and *Slice* 3 will be reduced significantly and the query execution will be accelerated. From this figure, we could see the time saved by intra-query fault tolerance directly. As another example, if a failure occurs after the checkpoint on *Op*.13, only *Slice* 4 will be executed during the rerun.

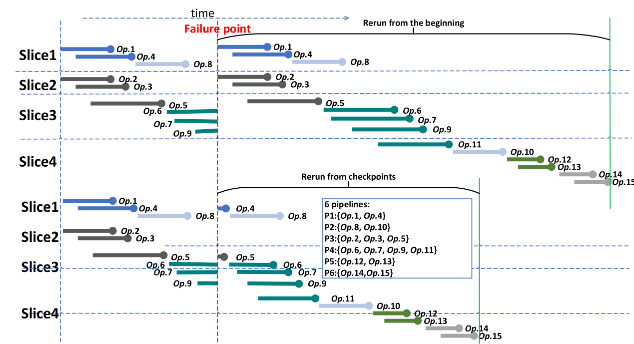

Fig. 5 A failure occurs during executing $Q17$. Before that, *Op*.4 and *Op*.5 are materialized. The query will rerun from intermediate results of *Op*.4 and *Op*.5 and Operators 1~5 will be skipped. *Slice* 2 will not be launched.

3.2 Selection of Optimal Checkpoints

In a query plan, many operators can be candidates of checkpoints. However, it is not necessary to materialize them all, which may introduce unnecessary overhead to query processing. As long as a desired success rate can be achieved, checkpoints should be as fewer as possible. Thus, we need to select an optimal set of checkpoints that incur the minimum materialization cost.

Databases usually estimate the cost of a query plan during query optimization. Based on their cost estimation, we can estimate overhead of checkpointing as well as success rate. This will lead us to an optimal checkpointing plan. For instance, the query plan in Fig. 3 offers 6 checkpoint candidates, i.e.,{*Op*.4, *Op*.5, *Op*.10, *Op*.11, *Op*.13, *Op*.15 }. A possible checkpointing plan is to activate any subsets of them, e.g., {*Op*.4, *Op*.11, *Op*.13, *Op*.15}. In total, there will be 2^6 combinations or 2^6 candidate checkpointing plans. For each candidate plan, we can use Monte Carlo method to estimate its success rate. Since the search space is exponential and Monte Carlo method is time-consuming, we employ some pruning methods to accelerate searching.

In general, SIFT's checkpointing plan optimization consists of 3 steps:

1. Collapsing the query plan and estimating cost of each candidate checkpointing plan (see Sect. 3.2.1);
2. Calculating success rate of each candidate checkpointing plan (see Sect. 3.2.2 for the computation model and Sect. 3.2.3 for the Monte Carlo method);
3. Enumerating all candidate plans to identify an optimal one (see Sect. 3.2.4).

3.2.1 Query Plan Collapsing and Cost Estimation

For a given checkpointing plan, we only need the query processing costs between checkpoints to calculate the success rate. For simplicity of analysis, we collapse the query plan that only retain checkpointing operators. Collapsing is performed twice for each query plan. First, it generates a collapsed query plan, P_C, which preserve checkpointing candidates only. Other operators are upwardly compressed into the checkpointing candidates. Based on P_C, it then generates a collapsed checkpointing plan, P_{MC}, in which only checkpoint operators remain. For example, the query plan in Fig. 3 can be collapsed into the one in Fig. 6a. In the collapsed plan, *Op*.1 is merged with *Op*.4 to form a new operator *Op*.*A*. The plan mentioned in Sect. 3.2 will be collapsed into the checkpointing plan in Fig. 6b.

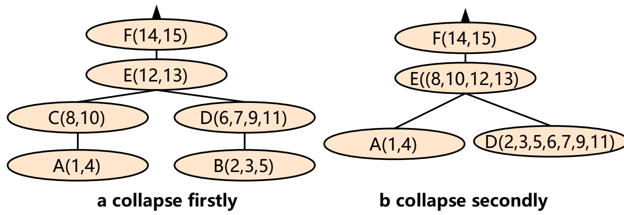

Fig. 6 Collapsed query plan of $Q17$

Cost estimation is performed on P_{MC}. The aggregated costs of the operators between adjacent checkpoints are calculated. For instance, cost of $Op.A$ in Fig. 6b will represent the cost or time consumed by the query processor before the checkpoint $Op.A$. If a failure occurs after $Op.A$, this cost or time will be saved during the rerun.

As I/O is usually the performance bottleneck, our cost model is based on I/O. For each $Op.i$, we estimate its execution cost Cr_i and materialization cost Cm_i. Cr_i represents I/O cost of executing $Op.i$. For an established database system, it is directly available in the query plan. Cm_i is the overhead of performing checkpointing on $Op.i$. In SIFT, Cm_i can be estimated based on Ci_i, which is I/O cost for materializing intermediate results of $Op.i$ to local storage. We calculate Cm_i as $Cm_i = Mf_i * Ci_i$, where Mf_i is a factor greater than 1, as checkpointing will incur extra cost of data transmission—rather than materializing intermediate results locally, checkpointing will transmit the intermediate results to the mirror node for materializing. We use Ct_i to represent the total I/O cost consumed by an operator under SIFT, that is, $Ct_i = Cr_i + Cm_i$. Then, the total cost of running a query can be calculated as,

$$t_{SIFT} = \sum_{i \in P_{MC}} Ct_i. \tag{3}$$

3.2.2 Success Rate Calculation

In Sect. 2.3, we showed how to calculate success rate without intra-query fault tolerance. In this section, we consider the case when intra-query fault tolerance is enabled. Given a time constraint T, the success rate $Succ(T)$ can be amended as follows:

$$\begin{aligned} &Succ_0(T) = e^{-\lambda t_{SIFT}} \\ &Succ_i(T) = (1 - Succ_{i-1}(T)) * P_i(t_{SIFT}, T),\ i \in 1, 2, \cdots \\ &Succ(T) = Succ_0(T) + Succ_1(T) + Succ_2(T) + \cdots + Succ_i(T) + \cdots \end{aligned} \tag{4}$$

In this equation, $P_i(t_{SIFT}, T)$ represents the probability that a query successfully finishes at the $(i+1)$th attempt, i.e., after i failures. Based on the assumption of exponential distribution, the timeline of the collapsed query plan in Sect. 3.2.1 can be illustrated by the first line in Fig. 7 (post-traveling of P_{MC}). Other lines illustrate the different situations of failures. By applying SIFT, the query will be rerun at the latest checkpoint. SIFT can shorten the reruns remarkably.

$P_1(t_{SIFT}, T)$ is a complex piecewise function. If it is unfolded, the direct calculation of $P_i(t_{SIFT}, T)$ will be infeasible. To make the estimation of success rate possible, we apply Monte Carlo method to compute $P_i(t_{SIFT}, T)$.

3.2.3 Application of Monte Carlo

The essence of the Monte Carlo method [12] is simulation. As the probability of one-time success might be high, it will be wasteful to simulate the situations when no failure occurs. Instead of estimating success rate directly, SIFT estimates the conditional probability given that at least one failure occurs. As an amendment of Eq. 4, our formula for estimating success rate is:

$$Succ(T) = e^{-\lambda t_{SIFT}} + (1 - e^{-\lambda t_{SIFT}}) * P(t_{SIFT}, T), \tag{5}$$

where $P(t_{SIFT}, T)$ presents the probability of a successful query execution after one failure occurs. SIFT employs Monte Carlo method to estimate $P(t_{SIFT}, T)$, based on which it calculates the success rate $Succ(T)$.

When running the Monte Carlo method, we use the random number r to represent the time interval before the next failure, and s_t to represent the start point of this simulation, which is initially 0. The simulation of query execution will be repeated until one of the following inequalities holds.

$$\begin{aligned} &a : t_{SIFT} < s_t + r \\ &b : T < s_t + r + n_f * MTTR \end{aligned} \tag{6}$$

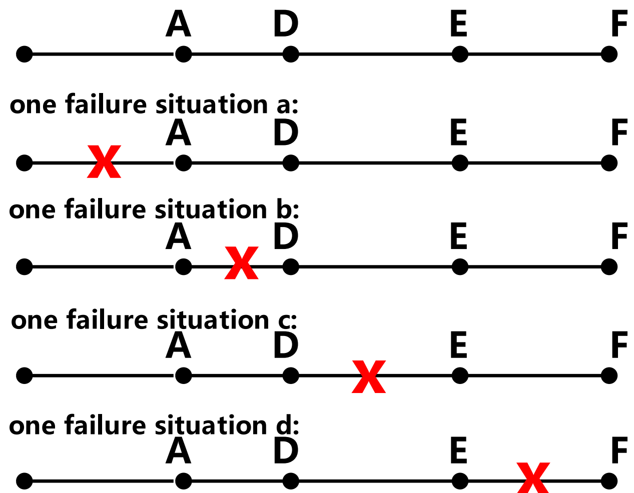

Fig. 7 Timeline of $Q17$ and situations of one failure

In Eq. 6, n_f represents the number of failures that have already occurred. *MTTR* represents the Mean Time To Repair, i.e., the time required to prepare the rerun. In each round of simulation, if Eq. 6.a holds, a successful case is recorded. Otherwise, a failed case is recorded. After a sufficient number of simulations, the conditional probability $P(t_{SIFT}, T)$ can be computed as the fraction of the succeeded cases.

3.2.4 Enumerating Plans

For a collapsed plan with n_c candidate checkpoints, there are 2^{n_c} possible checkpointing plans. The search space of checkpointing plans for a query can be structured as a tree. For instance, the search space of $Q17$ is illustrated in Fig. 8. SIFT performs Breadth-First Search over the tree to evaluate the plans. It employs a pruning method to avoid unnecessary work and accelerate the evaluation. In general, we set a target success rate SR_T. SIFT will stop the search on a branch if it has achieved SR_T. For instance, if checkpoints {A, B} has already satisfied SR_T, we would not consider set of checkpoints {A, B, C}, as {A, B, C} will only incur more materialization cost than {A, B}. If no candidate plan could achieve SR_T, SIFT chooses the plan with the highest success rate.

4 Implementation

We implemented SIFT in Greenplum [2], one of the most popular open-source MPP database systems. It adopts a scale-out and shared-nothing architecture and provides analytical capability on data of petabytes. Greenplum Version 4.3 contains more than 1.5 million lines of code in about 6000 files. To implement SIFT, about 7000 LOCs were added or modified. The main changes are shown in Fig. 9, and the detailed statistics can be found in Table 2.

4.1 Modification on the Master Node

The implementation on the master node includes the following four parts.

1. *Failure handling* Originally, Greenplum does not support intra-query fault tolerance. After a segment crashes, the system replaces the failed segment with its mirror. However, interrupted queries will be skipped. After recovered, the system will continue to serve following requests. To enable intra-query fault tolerance, we adjusted the behavior of master node—it will actively check for unfinished queries and rerun them automatically.
2. *Query re-execution* To rerun an unfinished query, the system needs to know which checkpoint to start from. The judgment should be made by the master and informed to the slaves. During query processing, each operator will record their states of checkpointing. When rerunning a query, the master will collect the states to identify the latest completed checkpoint by traversing the query plan tree. Then the operators succeeding the checkpoint will be activated and the ones preceding it will be skipped. For example, in Fig. 3, if *Op*.4 has been checkpointed, *Op*.1 could be skipped. If *Op*.13 is checkpointed, *Op*.10 and *Op*.11 will be skipped. If all operators in a slice are skipped, the slice will not be instantiated.
3. *Checkpoint selection* The module of checkpoint selection is implemented in the query optimizer on the master node. To measure the I/O cost of checkpointing, we can reuse the cost estimation module of Greenplum. The resulting checkpointing plan will be broadcasted to slaves along with the query plan.
4. *Transaction management* When rerunning queries, it is necessary to consider transactional correctness. Greenplum adopts multi-version concurrency control (MVCC) [13]. The order of each transaction is determined by its Transaction ID and Transaction Snapshot. Therefore, to ensure the correctness of a re-executed query, we can simply reuse the ID and Snapshot of the original transaction. This requires us to save the original ID and Snapshot on both master and slave nodes.

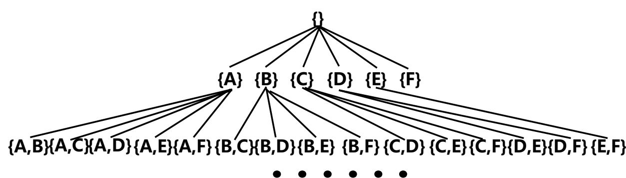

Fig. 8 Searching space of checkpointing plans for $Q17$

4.2 Modification on the Slave Node

The implementation on the slave side mainly aims to enable checkpointing. That includes materializing their intermediate results on mirrors and reusing them after failures. Table 2 shows that the implementation of the checkpointing module constitute a large part of the changes on the whole system. The module of checkpointing was implemented on operators of *hash*, *sort*, and *aggregation*.

1. *States of checkpoints* For each candidate operator, we need to maintain its checkpointing states. This allows us to judge whether an operator has been checkpointed and whether the checkpoint is complete when rerunning a

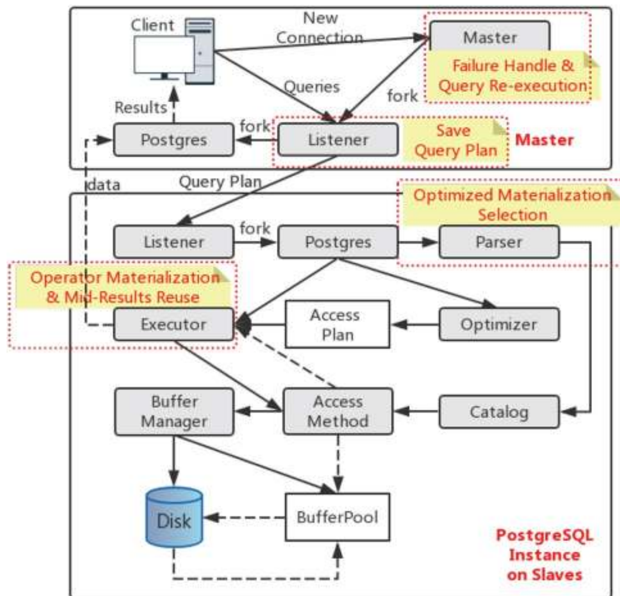

Fig. 9 Mainly changes for applying SIFT to Greenplum

Table 2 Code statistics for implementing SIFT on Greenplum database system

Model	Line of code
Optimized checkpointing selection	1800
Query restart	300
Redo point checking	300
Consistency guarantee	300
Intermediate result storage	600
Copy intermediate result to mirror	400
Intermediate result materialization	3300
Total	7000

query. For the former, we defined a label to identify the status of checkpoints. Its format is:

queryId_sliceId_operatorId_timestamp.

queryId identifies the query. Similarly, *sliceId* and *operatorId* are identifiers of the slice and the operator. *timestamp* is used to distinguish between the different rounds of execution of the same query, as more than one failure may occur. In addition, an integrity tag is used to detect unexpected media failure. During recovery, a salve needs to check the integrity tag to determine the usability of the intermediate results.

2. *Transmission of intermediate results* In the original Greenplum, intermediate results of breaking operators are materialized locally. The checkpointing module only needs to move them to the mirrors. It is obviously faster to transmit the data on the fly. If we do the transmission after all the intermediate results are generated, we have to read them from disk, which incurs extra I/Os. Greenplum uses a temporary file structure called *BFZ* to store intermediate results. *BFZ* utilizes an in-memory buffer. Intermediate results will first be written into the buffer. When the buffer is full, it is flushed to disks. To reduce overhead of checkpointing, we transmit data to the mirror when it is still in the buffer. This can save a significant amount of I/Os.
3. *Compression of intermediate results* To further reduce the overhead of intermediate result materialization, we perform compression. Compression is a common technology to reduce disk I/O in database, but it is rarely used for intermediate results by default. Greenplum provides a built-in compression module called zlib [14]. Instead of using zlib, we applied the compression library Zstandard [15], as it offers a better compression rate and a greater speed of compression and decompression. Moreover, it supports streaming compression, i.e., compress the file incrementally, which fits in our framework better.

5 Evaluation

We conducted experiments to evaluate SIFT. In this section, we show the effectiveness and overheads of SIFT, and compare it against some alternative approaches.

5.1 Experimental Setup

In the experiments, we used the TPC-H benchmark [11]. Its schema includes 8 tables. To show robustness of our strategy, we used two computer clusters of different scales, whose machine configurations are summarized in Table 3.

- *Cluster A* one master node and 16 slave nodes, with a data size of 500*GB*.
- *Cluster B* one master node and 8 slave nodes, with a data size of 200*GB*.

In the two clusters, each slave hosts two primary and two mirror segments. Data of each table is partitioned horizontally into segments and distributed to all slave nodes. Among the 22 queries of the TPC-H benchmark, we chose 6 typical queries to carry out our experiments. They include the 2nd, 7th, 9th, 16th, 17th, and 21st queries($Q2$, $Q7$, $Q9$, $Q16$, $Q17$, $Q21$).

Our experiments evaluated four systems:

- *orig* the original Greenplum (Version 4.3), which is not intra-query fault tolerant.
- *SIFT* the Greenplum system that implements SIFT.
- *materi-all* the Greenplum system that materializes all candidate operators of SIFT.
- *simulated-XDB* an simulated-XDB system [6] on Greenplum. It regards all operators as checkpointing candidates. (SIFT only considers breaking operators.)

5.2 Effects of Intra-query Fault Tolerance

Each of our experiments was performed for at least 20 times. Figure 10 shows the average success rates on the first three systems. Average success rates with 90% confidence intervals are shown in Fig. 10. And we can see from the figure, results are stable. For all the experiments, we set *MTBF* as 48h and time constraint T as 1.6 times of $t_0(1.6t_0)$. We ignored *MTTR* in the experiments, as it is much shorter than execution time of long queries. We can see that SIFT can promote the success rates significantly, especially for $Q17$ and $Q21$. These two queries' original success rates are relatively low, which makes the effects of fault tolerance more significant. They also offer more candidates for checkpointing, which enables us to find better checkpointing plans. On both clusters, we obtained similar results.

For some queries, such as $Q16$, the improvements made by SIFT are less visible. The execution times of the queries are shorter. Thus, their original success rates are already high and near to 100%, and little space is left for improvements. If we assume that at least one failure will occur during the query execution, the conditional probability will become the ones in Fig. 11. In this case, the effect of intra-query fault tolerance is amplified. For instance, SIFT can raise the success rate of $Q21$ from 24.4% to 76.6% on cluster A, and from 50.1% to 97.6% on cluster B. Intra-query fault tolerance will be more effective if the clusters are less stable.

We chose 4 queries, $Q7$, $Q9$, $Q17$, and $Q21$, to carry out further experiments, in which we manually shut down a slave node to see the impact of failures. Two rounds of experiments were performed for each query. In each round, one failure occurred, but at a different stage. Figure 12 shows results on cluster A, and Fig. 13 shows results on cluster B. In the figures, qi_j represents the jth round of test of the ith query; bt represents the time span between the start of the query and the failure; ct represents the time required to rerun a query when fault tolerance is enabled; et_c presents the time required to rerun a query when fault tolerance is disabled, i.e., the time needed for rerun a query with one failed node. et represents the time required to execute a query without failure. Usually, et_c is larger than et as there is one less node usable for executing the query. In the figure, the first bar represents the time a query required without failure. The second bar represents the time a query required if a failure occurs and fault tolerance is enabled. And the third bar represents the time a query required if a failure occurs and fault tolerance is disabled.

As Figs. 12 and 13 shows, $bt + ct$ is significantly smaller than $bt + et_c$. The former is the query time at the presence of one failure for SIFT. The latter is that value for the system without intra-query fault tolerance. This indicates that SIFT can effectively shorten the latency of query re-execution. Besides, we can see that ct drops as bt increases. This indicates that less work is needed to rerun a query when the failure is near to end of the query. This complies with our analysis in Sect. 1. That is, the later the failure occurs, the more work can be checkpointed. As the original system will waste what have already done before failures, the later the failure is, the more time it will waste. Thus, compared to original system, our approach could reduce the wasted time significantly while the failure is near to end of the query. As we can see, $bt + ct$ can be bigger than et. On the one hand,

Table 3 Configuration of the experimental machines

Operating system	Centos 7.3
Vendor_id	GenuineIntel
Address sizes	40 bits physical, 48 bits virtual
Disk throughput	700 MB/S
Number of cores	2
CPU (MHz)	1995.192
Cache size (KB)	4096
Memory (GB)	8
Disk (GB)	300

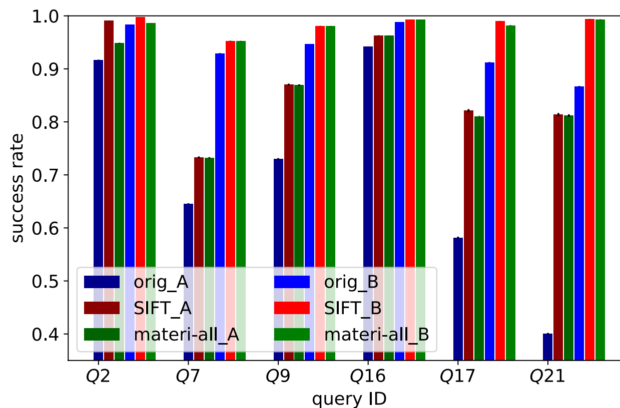

Fig. 10 Success rate of different queries on different clusters

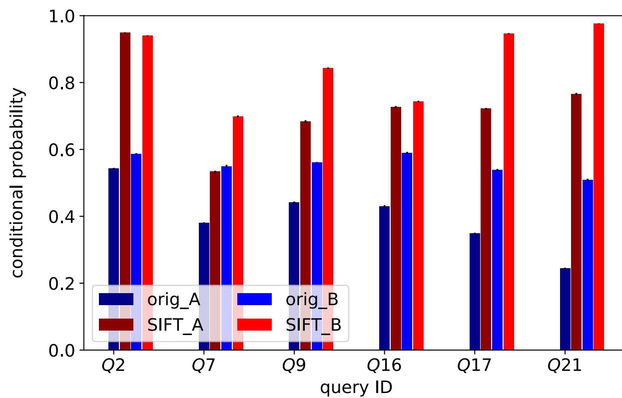

Fig. 11 Conditional probability under the condition of at least one failure

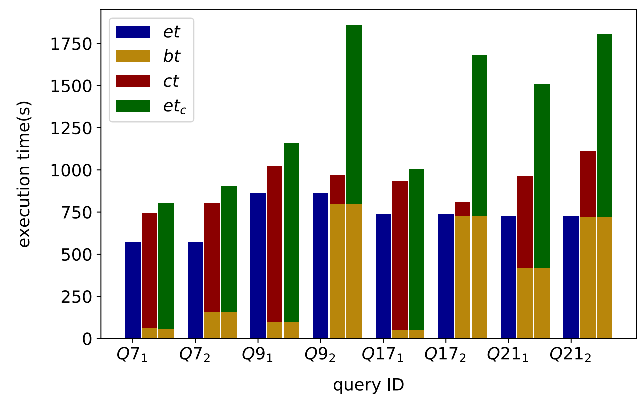

Fig. 13 Crash experiment on Cluster B

the work after the latest checkpoint is still wasted. On the other hand, after recovering, less slaves, i.e., less resources, are employed to execute the query, making the query processing slower. Figure 14 shows the ratio of time saved by intra-query fault tolerance, i.e., $\frac{et_c - ct}{bt + et_c - et}$. It represents benefits of using SIFT. As we can see, in most cases the ratio is more than 60%.

5.3 Overhead of SIFT

Overhead of checkpointing is inevitable. We tested all the four systems in Sect. 5.1 on cluster A, and the first two on cluster B, and recorded their query times. For the system, simulated-XDB, we measured the runtime when it attempted to achieve similar success rates as ours. Each of our experiments was performed for at least 10 times. We treat query time on original Greenplum system as 1.0, and normalize the query times of other systems based on it. This helps us see the differences. The average normalized execution time with 90% confidence intervals is shown in Fig. 15. As we can see, for most of the queries, the overheads of SIFT is visible. However, it is low most of the time. On average, the overhead introduced by SIFT is about 10% of the total execution time on cluster A and 8% on cluster B. As we mentioned before, intermediate results need to be materialized locally and remotely. And it is the main part of the overhead. In other words, the size of intermediate results impact the overhead significantly. This explains the higher overheads for some queries. In the architecture of Greenplum, mirrors and primaries reside on the same physical machines (Sect. 2.1). Thus, backups on mirrors will interfere with the original query execution and introduce significant overhead. However, this overhead could be easily reduced if we backup intermediate results somewhere else.

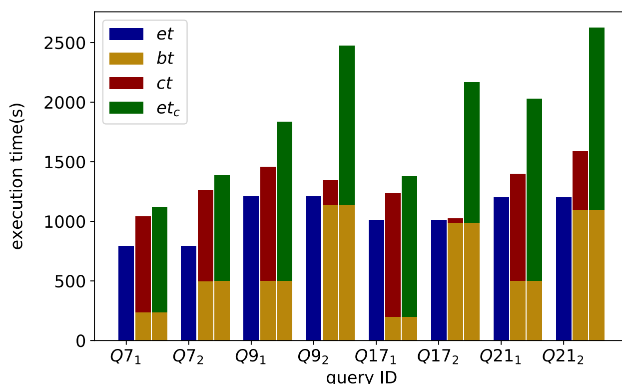

Fig. 12 Crash experiment on Cluster A

The results also indicate that it is not necessary to activate all candidate checkpoints, as it does not always benefits performance or success rates. In other words, optimized checkpoint selection can both reduce the overhead and improve the success rate. Table 4 shows the total number of candidate checkpoints and the number of checkpoints selected by SIFT. In our tests, as we regard success rate as our goal, materializing all candidates is usually not an ideal strategy. For instance, for $Q2$, if we do not perform checkpoint selection, the success rate is 94.86%. If we perform checkpoint selection, the success rate can rise to 99.08%. When the execution time is shortened, the probability of run-time failure can be naturally reduced (Sect. 2.3.2).

The experiment results on the simulated-XDB show that not all operators are good candidates for checkpointing. If a checkpoint breaks the query processing pipeline, it usually introduces a significant overhead. In this case, the system needs to materialize much more intermediate results, making the query time much longer. As we can see, SIFT remarkably outperforms XDB in quite a number of queries, as it never intends to break pipelines.

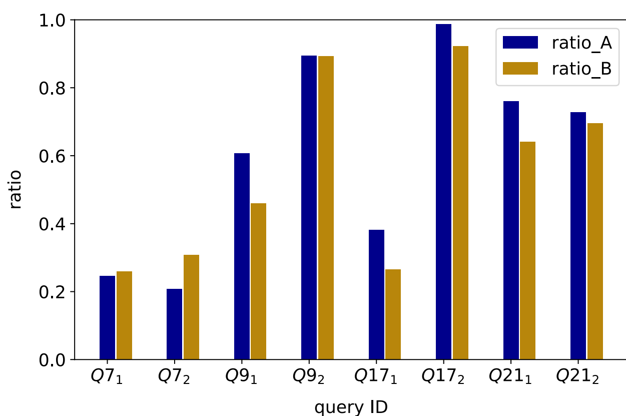

Fig. 14 Ratio of time saved by intra-query fault tolerance, i.e., $\frac{et_c-ct}{bt+et_c-et}$

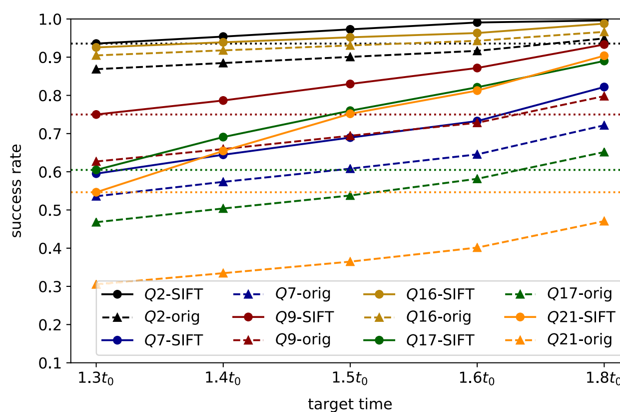

Fig. 16 Success rate changed with the given time on cluster A

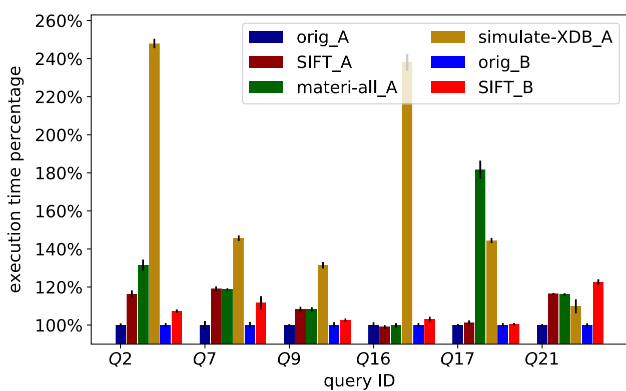

Fig. 15 Overhead of SIFT

5.4 Impacts of Parameters

In this set of experiments, we evaluated how different parameters affect the performance of SIFT. Firstly, we consider the influence of the time constraint T. Figure 16 shows the success rates when T is set as $T = 1.3t_0$, $T = 1.4t_0$, $T = 1.5t_0$, $T = 1.6t_0$ and $T = 1.8t_0$. *MTBF* was still set as $48h$. As expected, the larger the T, the higher the success rate. And for all of them, there is an increase on success rate compared with the original system. And Fig. 17 shows the conditional probability under the condition of at least one failure with different given time. For the queries $Q2$, $Q9$, $Q17$ and $Q21$, the success rates of system applying SIFT under the condition of $T = 1.3t_0$ are even higher than those of original Greenplum under the condition of $T = 1.6t_0$.

Another set of experiments were conducted to study the influence of *MTBF*, in which T was set to $1.6t_0$. The results can be found in Fig. 18. We can see that the drop of *MTBF* will increase the probability of failure and thus magnify the fault tolerance effect of SIFT. In other words, the benefits of SIFT will be more significant if *MTBF* is smaller. On the other hand, MTBF will drop as the scale of the cluster increases. Thus, SIFT is supposed to be more effective on larger clusters. Figure 19 shows how the conditional probability of at least one failure varies with *MTBF* . We can see that, on original system, the conditional probability will converge to 60%, as T is set to $1.6t_0$. For the system implementing SIFT, that value can be significantly improved. For instance, on the queries $Q2$, $Q17$, and $Q21$, it could approach 100%.

The results of $Q21$ on the two clusters are shown in Figs. 20 and 21 respectively. In the figures, *S_SR_C* means the success rate on the cluster C for the system S, and *S_CP_C* means the conditional probability with at least one failure on the cluster C for the system S. These results lead us to the same conclusion.

5.5 Effects of Compression

Although I/O operations are often seen as a performance bottleneck, CPU operations cannot be ignored. Compression can effectively reduce I/O operations. However, it comes with a price of extra CPU cost. In order to understand the

Table 4 Optimization strategy results

Qid	2	7	9	16	17	21
Candidate number	11	8	9	7	6	9
Checkpoint number	6	4	4	3	3	4

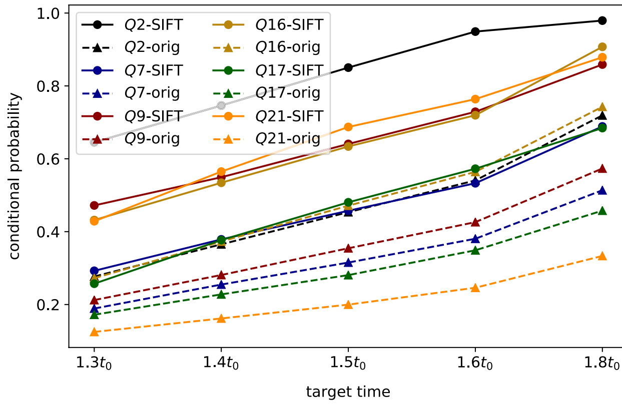

Fig. 17 Increase of success rate changed with the given time on cluster A

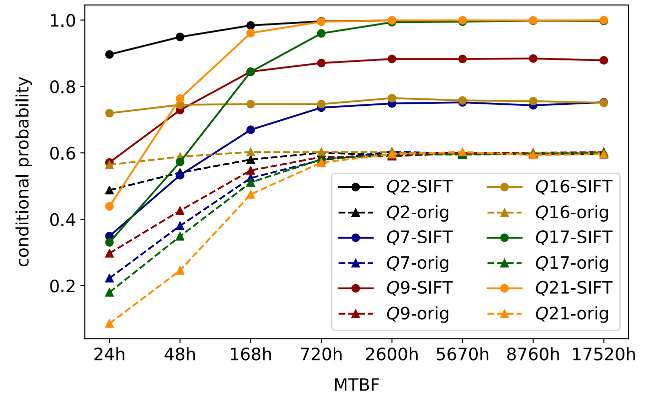

Fig. 19 Conditional probability changed with the MTBF on cluster A

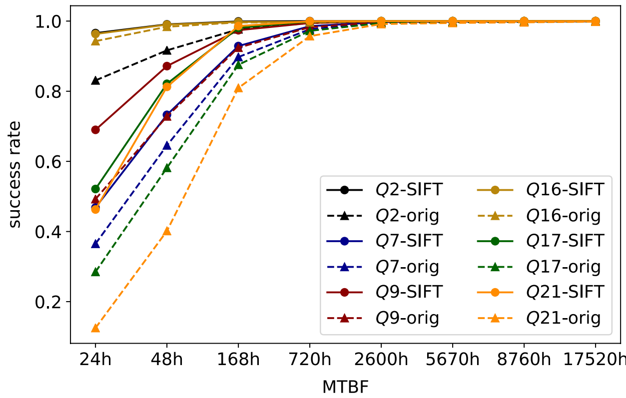

Fig. 18 Success rate changed with the MTBF on cluster A

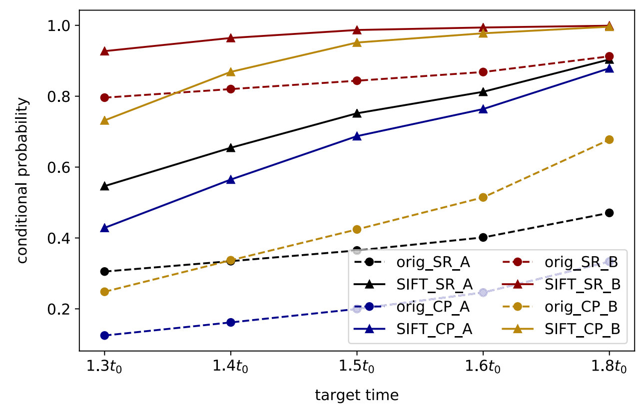

Fig. 20 Success rate of $Q21$ changed with the given time on the two clusters

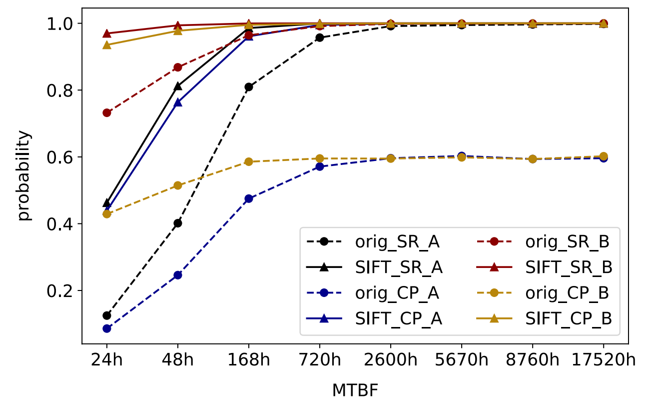

Fig. 21 Success rate of $Q21$ changed with the MTBF on the two clusters

influence of compression, we designed a simple query plan shown in Fig. 22. It comprises two *hashjoin* operators, $Op.3$ and $Op.8$, which we chose as checkpoints. On each segment, the average sizes of hash tables of $Op.3$ and $Op.8$ are $194MB$ and $757.5MB$ respectively. To speedup the checkpointing process, we perform compression on the hash tables. The compression ratios are 2.8 and 3.48 respectively. In other words, the data size after compression is about 30% of the original.

Figure 23 shows the time consumption of different stages of query processing. tc_i represents time required by compression in Slice i. td_i represents time required by decompression in Slice i. IO_i represents time for persisting the intermediate results before compression. IO_i^{comp} represents persistence time after compression. In the figure, the first bar measures the total time for materializing intermediate results with compression

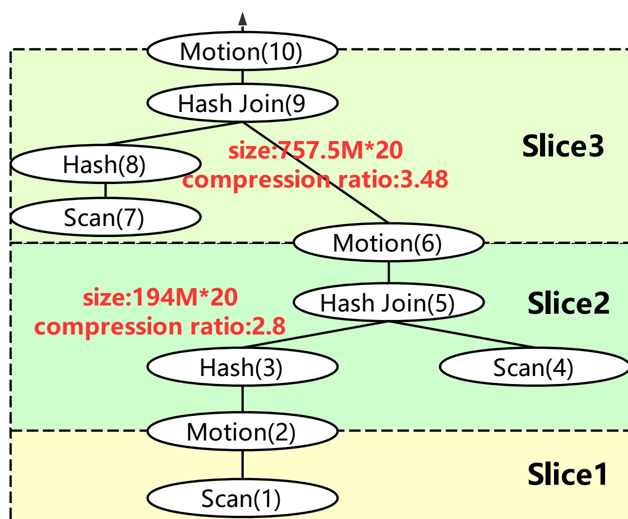

Fig. 22 Query processing plan of compression test query

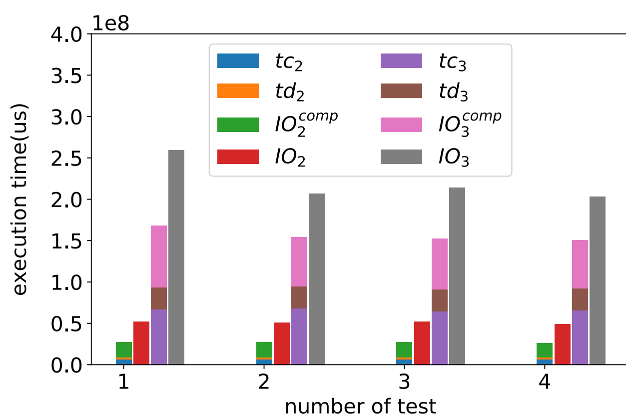

Fig. 23 Time consumption of different stages of query processing while using compressing

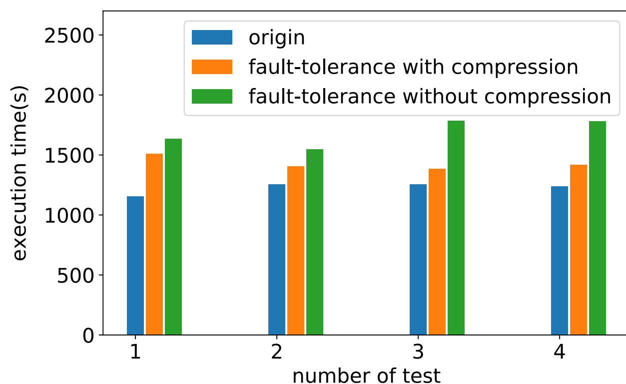

Fig. 24 Efficiency of compression

for *Slice*2, the second are measures that without compression. The other two bars represent those values of *Slice*3. We can see that compression is useful. Its effect is especially significant for Slice 2, in which a half of the time can be saved. Figure 24 shows the total execution time of the query in the four tests. It shows that compression can speedup query processing.

6 Related Work

6.1 Intra-query Fault Tolerance in Databases

Traditional relational databases [16–18] implement fault tolerance mainly through replication [19] and checkpointing at storage level. They usually do not take intra-query fault tolerance into consideration as queries on them usually could finish in seconds or minutes most of the time. There is little need for intra-query fault tolerance.

Well-known MPP DB products include Teradata [1], Greenplum [2], Vertica [3], as well as some SQL-on-Hadoop systems such as Impala [4] and HAWQ [5]. They all support fault tolerance at the storage level as the traditional databases do. To the best of our knowledge, none of them support intra-query fault tolerance.

The known work on intra-query fault tolerance includes FTOpt [10] and XDB [6]. XDB supports cost-based fault tolerance by choosing operators in a query plan to materialize. Its goal is to find a materialization configuration that would lead to minimum expected running time. However, XDB has little consideration about pipelines of query processing. It is prone to break pipelines of the original query and impair its performance. Besides, XDB is more complex to implement, as it needs to make every operators possible to materialize, which leads to re-implementation of all the query operators. We have shown in our experimental evaluation, SIFT tends to be more efficient as it always tries to preserve the query processing pipelines.

FTOpt considers pipelines and presents an extensible and heterogeneous fault tolerance framework. It also considers the differences between operators and applies different strategies to handle them. However, FTOpt assumes that data transmission over the network among operators is always order preserving. This makes FTOpt impractical for most MPP databases, which normally do not guarantee order preserving data transmission.

Neither XDB nor FTOpt was implemented in real MPP databases. In contrast, the design of SIFT emphasizes practicality. Our goal is a light weighted fault tolerance mechanism that can be applied to most real MPP databases. Besides, SIFT aims to ensure success rate, so that it adopts a unique approach to optimize checkpointing plans.

6.2 Intra-query Fault Tolerance in Big Data Platforms

Since Google published the work of MapReduce [20] and GFS [21], open-source data processing platforms such as Hadoop [7] have become popular. Multiple stacks of systems have been built on top of them. Some SQL-On-Hadoop implement intra-query fault tolerance by utilizing the fault tolerance function of Hadoop [7]. However they usually suffer from too much overhead. The work of [22] implemented pipeline within the Hadoop MapReduce framework. Experiments showed that its overhead of materialization is high. Osprey [23] applies the thought of MapReduce to MPP systems. It splits a analytical query over a star schema into short sub-queries. Its fault tolerance builds upon the data replication scheme of chained declustering [24]. However, it does not consider query success rate, making it difficult to achieve a good trade-off between fault tolerance and performance.

Microsoft has introduced a general-purpose, distributed execution engine, Dryad [25]. It executes queries over what is a communication graph. It allows data between operators to either be pipelined or materialized. However, it requires manual configuration and pushes the work of optimization to the user of the system. Bubble Execution [26] is a new query processing framework for interactive workloads at cloud scale introduced by Microsoft. It balances cost-based query optimization, fault tolerance, optimal resource management, and execution orchestration. It divides a query execution graph into a collection of query sub-graphs (bubbles), which is unit of scheduling and fault tolerance. Inside a bubble, tasks are connected via pipe channels. And between bubbles, tasks are connected via recoverable channels. While for SIFT, the unit of fault tolerance is divided by checkpoint operators, which are breaking operators.

Many stream processing systems implement fault tolerance [27–33] by redundant processing, checkpointing, and remote logging. They often allow weaker recovery guarantees in exchange for improved performance, but it is not allowed by databases. Some systems, such as Flink, provide exactly-once semantics. However, as a stream processing systems, they usually don't allow the existence of breaking operators, such as *hashjoin* and *sort*. The system proposed in [30] aims at find a materialization configuration with minimized overhead. But it considers only one failure.

In summary, most big data platforms, such as Hadoop, Spark and Flink, provide fault tolerance support at the data processing level. OLAP engines over these platforms can take advantage of it. While the existing OLAP engines still face performance issues, they can hopefully be improved as more optimization techniques are introduced. Nevertheless, SIFT is designed for traditional MPP databases rather than big data platforms.

7 Conclusions

In summary, we made the following contributions in this paper:

1. We proposed a smart intra-query fault tolerance (SIFT) mechanism for MPP databases, which can achieve a good trade-off among *fault tolerance effect*, *performance*, and *implementation cost*. It helps MPP database improves the success rates in processing large scale OLAP queries.
2. SIFT provides an optimization algorithm, which chooses the most appropriate set of checkpoints for a given query plan, to promote the query success rate and boost the performance of query processing.
3. We demonstrated the effectiveness of our techniques through an implementation of SIFT in a real-world MPP database, Greenplum. Experiments on TPC-H benchmark showed SIFT allows the system to achieve improved query success rates at the minimum price of intermediate result materialization. Nevertheless, further experiments and optimizations may be taken in the future to confirm the generality of the work in a wider range of scenarios.

Acknowledgements Thanks to WHC, ZYZ, ZJW, WCW who helped implement the idea. This work is partially supported by the NSFC Project No. 61772202.

Author Contributions ZX and CYP guided this research and proposed the idea. JYH, QIY and RLP specified and implemented the idea.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Teradata. https://www.teradata.com/
2. Greenplum. http://greenplum.org/
3. Vertica. https://www.vertica.com/
4. Apache Impala. https://impala.apache.org/
5. Apache HAWQ. http://hawq.incubator.apache.org/
6. Salama A, Binnig C, Kraska T, Zamanian E (2015) Cost-based fault-tolerance for parallel data processing. In: Proceedings of the

2015 ACM SIGMOD international conference on management of data, SIGMOD'15. ACM, New York, NY, USA, pp 285–297
7. Apache Hadoop. http://hadoop.apache.org/
8. Apache Spark. https://spark.apache.org/
9. Cant T, Mahony B, McCarthy J, Vu L (2006) Hierarchical verification environment. In: Proceedings of the 10th Australian workshop on safety critical systems and software—vol 55, SCS'05, Darlinghurst, Australia. Australian Computer Society, Inc, Australia, pp 47–57
10. Upadhyaya P, Kwon Y, Balazinska M (2011) A latency and fault-tolerance optimizer for online parallel query plans, pp 241–252
11. TPC-H Benchmark. http://www.tpc.org/tpch/
12. Raychaudhuri S (2008) Introduction to Monte Carlo simulation. In: Proceedings of the 40th conference on winter simulation, WSC'08. Winter simulation conference, pp 91–100
13. Bernstein PA, Goodman N (1981) Concurrency control in distributed database systems. ACM Comput. Surv. 13(2):185–221
14. Zlib. https://zlib.net/
15. Zstd. http://facebook.github.io/zstd/
16. Bartkowski S et al (2018) High availability and scalability guide for Db2 on Linux, UNIX and Windows. http://www.redbooks.ibm.com/redbooks/pdfs/sg247363.pdf. Accessed 4 Mar 2012
17. Bartkowski S et al. (2018) Oracle data guard. https://docs.oracle.com/cd/B19306_01/server.102/b14239.pdf. Accessed 4 Mar 2008
18. Microsoft. High availability solutions (SQL server). https://technet.microsoft.com/en-us/library/ms190202(v=sql.110).aspx. Accessed 4 Mar 2018
19. Hsiao H-I, Dewitt DJ (1993) A performance study of three high availability data replication strategies. Distrib Parallel Databases 1(1):53–79
20. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1):107–113
21. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. In: Proceedings of the nineteenth ACM symposium on operating systems principles, SOSP'03. ACM, New York, NY, USA, pp 29–43
22. Condie T, Conway N, Alvaro P, Hellerstein JM, Elmeleegy K, Sears R (2009) Mapreduce online. Technical report UCB/EECS-2009-136, EECS Department, University of California, Berkeley
23. Yang CM, Yen CY, Tan CC, Madden S (2010) Osprey: implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In: ICDE, pp 657–668, 11
24. Hsiao H-I, DeWitt DJ (1990) Chained declustering: a new availability strategy for multiprocessor database machines. In: Proceedings of the sixth international conference on data engineering. IEEE Computer Society, Washington, DC, USA, pp 456–465
25. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2007 Eurosys conference. Association for Computing Machinery, Inc, Lisbon, Portugal
26. Yint Z, Sun J, Li M, Ekanayake J, Lin H, Friedman M, Blakeley JA, Szyperski C, Devanur NR (2018) Bubble execution: resource-aware reliable analytics at cloud scale. Proc VLDB Endow 11(7):746–758
27. Hwang J-H, Balazinska M, Rasin A, Cetintemel U, Stonebraker M, Zdonik S (2005) High-availability algorithms for distributed stream processing. In: Proceedings of the 21st international conference on data engineering, ICDE'05. IEEE Computer Society, Washington, DC, USA, pp 779–790
28. Flink. https://ci.apache.org/projects/flink/flink-docs-release-1.4/internals/stream_checkpointing.html
29. Tatbul N, Ahmad Y, Çetintemel U, Hwang J-H, Xing Y, Zdonik S (2008) Load management and high availability in the borealis distributed stream processing engine. Springer, Berlin, pp 66–85
30. Li H, Wu J, Jiang Z, Li X, Wei X (2017) Minimum backups for stream processing with recovery latency guarantees. IEEE Trans Reliab 66:783–794
31. Carbone P, Ewen S, Fóra G, Haridi S, Richter S, Tzoumas K (2017) State management in apache flink®: consistent stateful distributed stream processing. Proc VLDB Endow 10(12):1718–1729
32. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M (2013) Naiad: a timely dataflow system. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles, SOSP'13. ACM, New York, NY, USA, pp 439–455
33. Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, Bhagat N, Mittal S, Ryaboy D (2014) Storm@twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data, SIGMOD'14. ACM, New York, NY, USA, pp 147–156