OTPM: Failure Handling in Data-intensive Analytical Processing

Binh Han*, Edward Omiecinski†, Leo Mark‡ and Ling Liu§
School of Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332
*binhhan@gatech.edu
†edwardo@cc.gatech.edu
‡leomark@cc.gatech.edu
§lingliu@cc.gatech.edu

***Abstract*—Parallel processing is the key to speedup performance and to achieve high throughput in processing large scale data analytical workloads. However, failures of nodes involved in the analytical query can interrupt the whole process, resulting in the complete restart of the query if the system does not have query fault-tolerance. Complete restart might be too costly for processing query on very large databases and might not be able to meet the time constraints in decision support systems. In this paper, we present an approach to resume query processing after failure by keeping track of the point at which data has been processed by an operator, called *operator tracking*. We also consider saving intermediate results using *partial materialization*. We look at several fundamental parallel database techniques which are widely used today and analyze the performance cost of query processing and recovery using those techniques with our OTPM fault-tolerance approach. We perform simulation-based experiments which show that our approach incurs only a small resume overhead compared to complete pipelining and complete materialization of intermediate results. Also, the combination of our approach with vertically partitioned database in a shared-nothing environment yields the best performance among different settings for parallel processing of data intensive analytical workloads.**

***Index Terms*—Failure handling, fault tolerance, analytical processing, parallel query processing.**

I. INTRODUCTION

Data volume is growing exponentially in data warehousing. Today, it is not unusual to hear about data warehouses with hundreds of terabyte size databases. Large commercial entities such as eBay, Yahoo and Facebook have already reached databases with sizes larger than a petabyte [1]. Data analysis involving large databases is often parallelized across nodes in large clusters in order to achieve high performance. However, this also increases the probability of failure during query processing as the number of nodes in the cluster increases, along with the trend to use low-priced hardware to carry out nontrivial workloads. Currently, there are two main approaches in parallel query processing: parallel DBMSs and the MapReduce-based systems [2]. Parallel DBMSs (PDBMSs) support fault tolerance at the granularity of a transaction, which only benefits queries that make changes to the database, not the read-only queries as in most analytical workloads. In a parallel DBMS, a query is executed in a pipeline of operators so that we have to recompute everything if we want to get access to the previous intermediate tuples. Consider a read-only query on a very large database that has been running for a long time and is about to finish. If a node fails, it will trigger the entire query to be reprocessed, which is sometimes unacceptable. In contrast to the complete pipeline execution of PDBMSs, the MapReduce approach to query fault tolerance is complete materialization. All intermediate outputs from an operator are locally saved to disk before executing the next operator. When a node goes down, its work can be restarted on an alternate node. However, performing complete materialization also degrades the performance of the whole process due to high I/O cost. In fact, it is one of the reasons why Hadoop and HadoopDB, two open source versions of the MapReduce framework, run significantly slower than two parallel DBMSs, Vertica and DBMS-X [3, 4].

In this paper, we present an efficient query fault tolerance scheme which avoids both complete restart and complete materialization, supports partial query restart by using operator tracking and partial materialization. The rest of the paper is organized as follows. In section II, we describe our platform assumptions. Our approach is then described in detail in section III. We present an analytical evaluation of our approach compared to complete pipelining and complete materialization in section IV. The experimental results are shown in section V. Section VI contains related work and the paper concludes in section VII.

II. PLATFORM ASSUMPTIONS

In this section, we look at several fundamental parallel database techniques which are widely used today and pick the most popular techniques to use in our analytical model.

A. Parallel Architectures

There are three common parallel architectures: shared-memory, shared-disk and shared-nothing [5]. In a shared-memory system, all the processors share one memory and a set of disks. Its benefits are ease of programming and load balancing. However, memory contention is a problem in shared-memory systems because the memory bandwidth will become a bottleneck as the number of CPUs increases. In a shared-disk system, every processor has access to a collection

COLLABORATECOM 2011, October 15-18, Orlando, United States
Copyright © 2012 ICST
DOI 10.4108/collaboratecom.2011.64

of hard disks but has its own private memory. The shared-disk architecture also limits the scalability of the system as in the shared-memory approach. In contrast, each processor in a shared-nothing system has its own memory and disks. Every processor performs a task locally on its own resources which avoids the problem of resource contention. Therefore, of three approaches, shared-nothing systems scale the best [6]. The shared-nothing architecture is suitable for processing large data warehouse workloads. We assume a shared-nothing architecture for our analytical model.

B. Forms of Parallelism

Three forms of parallelism can be exploited in processing data analytical workloads [7]. First, we have the parallel execution of multiple queries, called inter-query parallelism. Next, there is the parallel execution of multiple operations in a query, known as intra-query parallelism. Lastly, we have the parallel execution of multiple sub-operations for a single operation, called intra-operation parallelism. We will use intra-query parallelism and intra-operation parallelism in processing our example query.

C. Data Placement Strategies

In parallel DBMSs, a relation is partitioned across the data sites usually horizontally or vertically [8]. In horizontal partitioning, the rows in a relation are spread across a number of data nodes by hashing, round-robin partitioning or range partitioning. In vertical partitioning, a relation is divided across data sites by projecting over its columns. After using horizontal or vertical partitioning, each data node has a fragment of the original relation. Each fragment can also be replicated. If two physical copies of a fragment (one primary fragment and one backup fragment) are stored on two different data nodes, it is called chain replication. If the system has N data nodes, the primary fragment is stored on one data node and its backup copy is subdivided into $N-1$ subfragments stored on $N-1$ other data nodes, which is known as interleaved replication.

The physical layout of the database is an important factor that affects load balancing and query optimization. Hence we consider both horizontal and vertical approaches.

III. Our Approach

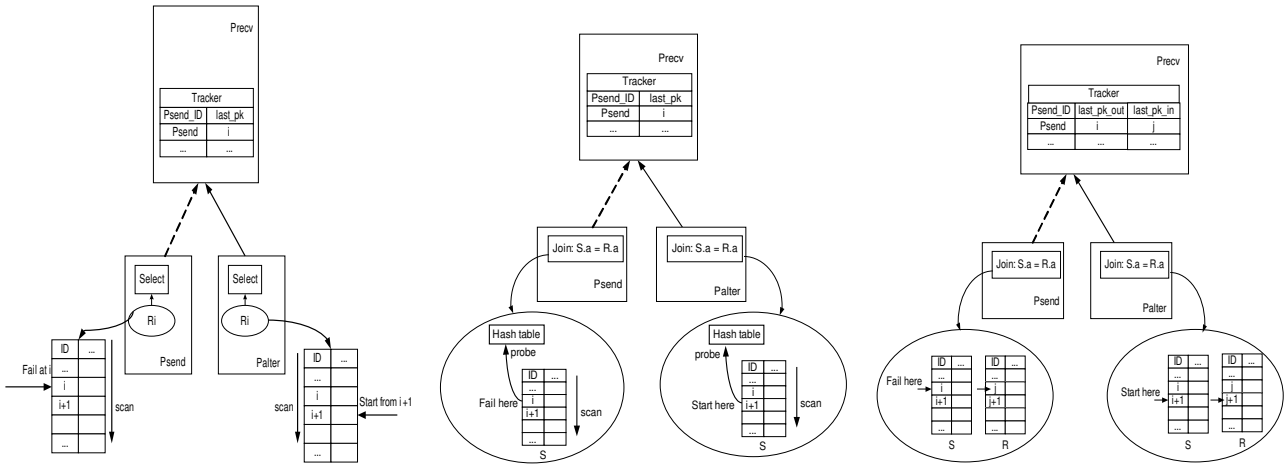
Suppose we have an SQL query that needs to be parallelized across multiple nodes in a cluster. The query optimizer will generate the best query plan based on the minimum estimated cost. A query plan is expressed as a tree of algebra operators. Each operator is assigned to nodes in the cluster by exploiting possible forms of parallelism. Each node is considered as a possible single point of failure. We call the nodes which receive data from a node P_i the *downstream* nodes of node P_i. The nodes sending data to a node P_i are called the *upstream* nodes of node P_i. As in Figure 1(a), P_{send} is the upstream node of P_{recv} and P_{recv} is the downstream node of P_{send}. Our approach prevents the loss of produced data and ensures that the query can be resumed from the point it stops in the event of failure. No duplicate tuples will be produced during failure recovery. In order to do this, an *operator tracker* is placed at every node to checkpoint the progress of its upstream nodes. In addition, resulting tuples produced by each node are cached locally in memory and are materialized on a backup node only upon exceeding memory. Using operator tracking requires that data is processed and produced in a deterministic order. Since every database table usually has a clustered index on its primary key, we can process and produce data in the order of the primary key. If the table also has an index on the select/ project/ aggregation/ join attribute, we will use that index instead of the index on the primary key. Without loss of generality, we explain our approach assuming data is processed and produced in the order of the primary key.

A. Operator Tracking

An operator tracker is placed at every node P_{recv} to checkpoint the progress of its upstream node P_{send}. The operator tracker stores different information corresponding to different types of operators (select/ project, join, aggregation) which P_{send} performs.

1) Select/ project: A select/ project scan usually performs directly on the data source since it is often pushed down as far as possible in the optimal query plan. Thus, we can take advantage of the index scan to process and produce resulting tuples in the order of the primary key. The operator tracker at P_{recv} will log the $last\_pk$, the primary key of the most recent tuple which P_{recv} has received from P_{send}, as described in Figure 1(a). The operator tracker keeps information in a temporary table in memory which has two fields: P_{send}_ID and $last\_pk$. Each row in this table is the checkpoint of each upstream node sending tuples to P_{recv}. When a new tuple reaches P_{recv}, the operator tracker at P_{recv} will update the $last\_pk$ value with the primary key of the new tuple. If P_{send} fails, its work will be taken over by an alternate node P_{alter}. P_{alter} is chosen from the sites which have a copy of the data stored on P_{send}. All previous connections to P_{send} now switch to P_{alter}. The checkpoint information $last\_pk$ kept at P_{recv} is sent to P_{alter}, telling P_{alter} to start scanning its input from the tuple next to the tuple whose primary key equals $last\_pk$, called the $last\_pk$ tuple from now on. This guarantees that no duplicate results are produced and sent out to P_{recv}.

2) Aggregation: To exploit intra-operator parallelism, an aggregate operator at each node is usually hash-based. Input tuples are hashed on the Group By attributes. Each row in the hash table will keep updating the accumulated result as in the aggregation function until the entire set of input tuples have been processed. Since hash-based aggregate cannot produce results until it has seen the entire input, P_{send} cannot transmit results to P_{recv} until it finishes processing. If P_{send} goes down before that time, a complete restart of the work of P_{send} at P_{alter} is required. If P_{send} has transmitted part of the results to P_{recv} when it fails, the tracker at P_{recv} keeps all the hash values of the Group By attribute that have been sent to P_{recv}. During failure recovery, P_{alter} can skip computing those tuples whose hash values are saved at P_{recv}'s tracker.



(a) Select (b) Hash Join (c) Nested Loop Join

Fig. 1. Operator tracking

3) Join: If there is a join of two relations in the SQL query, the query optimizer might implement a parallel hash join or a parallel nested loop join depending on the statistics of the data sources. First, the data sources are redistributed across the nodes. Then each node can do a hash join or a nested loop join and can produce the matching tuples in parallel. Suppose P_{send} performs a join on two operands S and R.

Figure 1(b) shows the operator tracker for a hash join. If P_{send} performs a hash join, then assume that R is the build operand which is used to build the hash table and S is the probe operand whose join attribute values are probed the hash table to find matches. The probe operand is scanned in the order of its primary key. Each matching tuple is tagged with the primary key of the probe operand that was used to produce the match before sending it downstream. The tracker on P_{recv} , like in the select/ project tracker described above, also keeps a table with two field (P_{send}_ID, $last\_pk$) rows, where $last\_pk$ is the tag of the most recent tuple transmitted to P_{recv}. This $last\_pk$ value is used, in case P_{send} fails, to tell P_{alter} to start scanning the probe operand at the tuple next to the tuple whose primary key equals $last\_pk$.

For a nested loop join, assume that S is the outer operand which is scanned one time and R is the inner operand which is scanned as many times as S's cardinality. Every matching tuple is now tagged with two primary keys, one for the tuple from S and one for the tuple from R used to produce the match, and then is sent downstream. The tracker on P_{recv} keeps a table with three field (P_{send}_ID, $last\_pk\_out$, $last\_pk\_in$) rows, where $last\_pk\_out$ and $last\_pk\_in$ represent the tag of the most recent tuple transmitted to P_{recv}. During recovery, P_{alter} continues the iteration through S at $last\_pk\_out$ tuple and search for a match through R from the tuple after $last\_pk\_in$ tuple, as illustrated in Figure 1(c). The next iteration is then performed as usual nested loop join.

For all types of operators, we can see that the information kept at each operator tracker is small and can certainly be kept in memory. Hence, the cost to maintain an operator tracker at each node is negligible.

B. Partial Materialization

Using operator tracking is not enough to restart partially at P_{alter} after failure. The operator tracker only checkpoints the progress running at P_{send} and informs P_{alter} to continue processing where P_{send} left off. One issue is how to supply the input tuples for P_{alter} to process. For operators that compute directly on relations stored on disks, i.e. operators that process the leaves of the query tree, P_{alter} will be referred to the data source sites to get its input during failure handling. But if the operator does not perform directly on the available database and intermediate results have not been saved, restarting at P_{alter} can cause all the upstream nodes to restart in order to produce data input for P_{alter}. One solution is to completely materialize intermediate results. However, complete materialization might be too costly to perform as it increases both total running time and failure rate. Hence, it is reasonable to perform partial materialization during query processing. Intermediate tuples are locally cached in memory and materialization is only done when memory is exceeded.

During query planning, several nodes in the system will be assigned for storing intermediate results when needed. Those nodes only act as storage nodes so the number of query sites in the system will decrease. When the memory at the operation node is full, we free it up by sending all the tuples in memory to the assigned backup node (log in the memory of the backup node if there is enough space or write to disk otherwise). Extra network traffic is incurred to transmit tuples from the operator node to the backup node. However, doing materialization on a backup node instead of the same operator node will not degrade the performance of the operator node. If a node fails, we can use the results from the previous level, logged in memory of the right upstream node and maybe on disk/ memory from a backup node if there's any to begin recovering at the point of failure.

In the rest of this paper, we will call our approach OTPM,

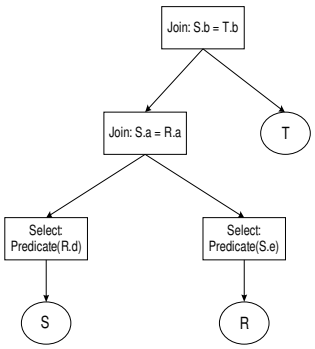


Fig. 2. A typical query tree.

which is the abbreviation of operator tracking and partial materialization. In addition, CP stands for Complete Pipelining and CM stands for Complete Materialization. We also use the terms site and node interchangeably in the paper.

IV. Analytical Cost Model

In a parallel DBMS, the database can be partitioned horizontally or vertically. A query is executed differently for different data placement strategies. Suppose we have a query tree as in Figure 2. We will describe the query execution and analyze the running cost of this query on a horizontally partitioned database and on a vertically partitioned database. Since the query is parallelized, the cost in terms of the time of running a task across multiple nodes is measured by the cost of the node which takes the maximum time. There have been several approaches for progress indicators (PIs) for SQL queries which estimate the amount of the work completed and the remaining time to complete SQL queries. In our paper, we consider failure at each operator node so we would like to apply one of those PIs approaches to determine the point of failure happening at each node. Here, we choose the points of failure with regard to the percentage of input tuples that has been processed at each node, as in [9]. Our evaluation is based on the TPC-H benchmark [10], the most widely used benchmark to measure performance of decision support systems involving large amount of data. It has a set of ad-hoc queries which models the activities of a business information analysis in a wholesale supplier. We use the TPC-H schema in Figure 3 with the scale factor SF=1 as the test data. The partitions for different relations are assumed to be stored on different data nodes. We take the following query, which is an example for the query tree above, to calculate the specific I/O costs and network costs.

SQL command:

```
SELECT T.* FROM part as R,
partsupp as S,supplier as T
WHERE T. suppkey = S.suppkey
AND R.Partkey = S.Partkey
AND S.ps_availqty < 1000
AND R.p_size < 21
```

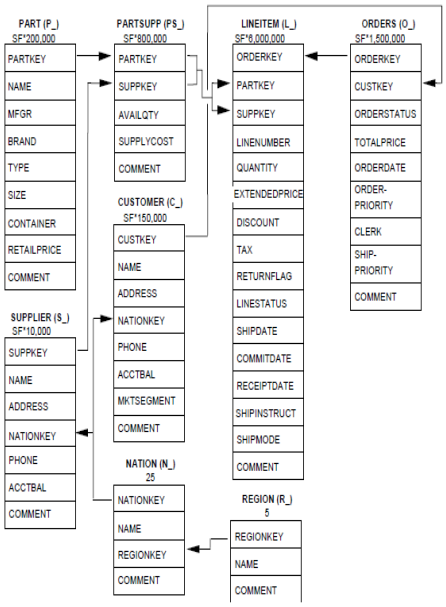


Fig. 3. The TPC-H database [10].

TABLE I
PARAMETERS USED IN THE ANALYTICAL COST MODEL

Sym.	Description	Values
t_{IO}	Time for a block access	3500 μs
t_{tup}	Time to iterate through a tuple	0.065 μs
t_{Blk}	Time to iterate through a block	0.020 μs
t_{func}	Time to call a function	0.009 μs
B	Block size	32KB
P	Network packet size	32KB
t_p	Time to transmit a packet	320μs[1]
R	Size of relation R	-
$\|R\|$	Number of tuples in relation R	-
ssl_R	The selectivity[2] of the select operator performed on relation R	-
jsl_R	The selectivity of the join operator performed on relation R	-

[1]With 100MB/s bandwidth
[2]The ratio of the output and input cardinalities

In this SQL query, S=partsupp, R=part, T=supplier. The selectivity of the predicate for Part (p_size $<$ 21) is 0.4. The selectivity of the predicate for Partsupp (ps_availqty $<$ 1000) is 0.1. The first join (partsupp $\bowtie$ part) produces 9624 tuples. The second join (partsupp $\bowtie$ supplier) produces 9624 tuples as well. Table I includes parameters used for calculating specific costs in our example. Some of the values are taken from [11].

A. For a Horizontally Partitioned Database

Each base relation in the query is equally partitioned across a number of data nodes. Relations R, S, T are spread over m, n, k data sites respectively. Figure 4(a) shows the optimized query plan, including three separate steps:

1) Do the select scan simultaneously on m partitions of R and n partitions of S then hashes the resulting tuples on the value of attribute a into u query sites A_1, A_2,

$\ldots$, A_u. Let R_i be the partition on which it takes the maximum time to perform these tasks. The cost to perform step 1 includes:

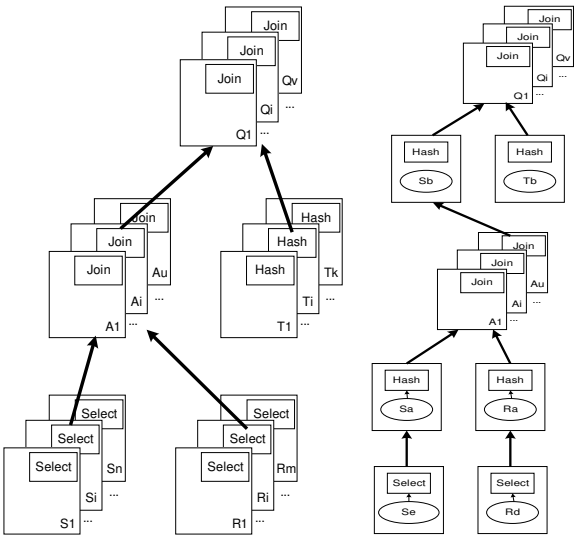
- Scan the partition: I/O cost = $\frac{R_i}{B} \times t_{IO}$
- Iterate through blocks, iterate through tuples in each block to check the select predicate, apply the hash function to each satisfying tuple:
  CPU cost = $\frac{R_i}{B} \times t_{Blk} + |R_i| \times (t_{tup} + t_{func}) + ssl_R \times |R_i| \times t_{func}$
- Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used):
  I/O cost = $2 \times \frac{(ssl_R \times R_i)}{B} \times t_{IO}$
- Send resulting tuples to the storage node (if OTPM is used): NW cost = $\frac{(ssl_R \times R_i)}{P} \times t_p$
- Send resulting tuples to query sites: NW cost = $\frac{(ssl_R \times R_i)}{P} \times t_p$

2) Let R_{A_j} and S_{A_j} represent the set of records sent to A_j ($j = 1, 2, \ldots, u$) and stored in temporary tables at A_j. Each query site A_j performs the hash join of R_{A_j} and S_{A_j}. The matching tuples are hashed on the value of attribute b then sent to v query sites $Q_1, Q_2, \ldots, Q_v$. Relation T is also repartitioned by the same hash function onto the same v query sites $Q_1, Q_2, \ldots, Q_v$.

- Write and read temporary tables:
  I/O cost = $\frac{(R_{A_j} + S_{A_j})}{B} \times t_{IO} \times 2$
- Iterate through blocks, iterate through tuples in each block to build the hash table (for the tuples in the smaller table) and probe for matches (for the tuples in the other table), apply the hash function to each matching tuple:
  CPU cost = $\frac{(R_{A_j} + S_{A_j})}{B} \times t_{Blk} + (|R_{A_j}| + |S_{A_j}|) \times (t_{tup} + t_{func}) + jsl_{S_{A_j}} \times |S_{A_j}| \times t_{func}$
- Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used):
  I/O cost = $2 \times \frac{(jsl_{S_{A_j}} \times S_{A_j})}{B} \times t_{IO}$
- Send resulting tuples to the storage node (if OTPM is used): NW cost = $\frac{(jsl_{S_{A_j}} \times S_{A_j})}{P} \times t_p$
- Send resulting tuples to query sites: NW cost = $\frac{(jsl_{S_{A_j}} \times S_{A_j})}{P} \times t_p$

3) Let S_{Q_i} and T_{Q_i} represent the set of tuples, which originally belong to relations S and T respectively, that have been sent to $Q_i (i = 1, 2, \ldots, v)$. Each query site Q_j will perform the join $S_{Q_i} \bowtie T_{Q_i}$ using the same algorithm as in the previous join of R and S. The matching tuples are sent to a central node which stores the final result of the given query.

- Write temporary tables and load them into memory:
  I/O cost = $\frac{T_{Q_i} + S_{Q_i}}{B} \times t_{IO} \times 2$
- Iterate through blocks, iterate through tuples in each block to build the hash table (for the tuples in the smaller table) and probe for matches (for the tuples in the other table): CPU cost = $\frac{T_{Q_i} + S_{Q_i}}{B} \times t_{Blk} + (|T_{Q_i}| + |S_{A_j}|) \times t_{tup} + t_{func}$
- Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used):
  I/O cost = $2 \times \frac{(jsl_{S_{Q_i}} \times T_{Q_i})}{B} \times t_{IO}$
- Send resulting tuples to the storage node (if OTPM is used): NW cost = $\frac{(jsl_{S_{Q_i}} \times T_{Q_i})}{P} \times t_p$
- Send the final result to a central node:
  NW cost = $\frac{(jsl_{S_{Q_i}} \times T_{Q_i})}{P} \times t_p$



(a) Horizontally partitioned (b) Vertically partitioned

Fig. 4. The query plan for partitioned database.

B. For a Vertically Partitioned Database

For vertical partitioning, relation R, which has m attributes, is partitioned into m data nodes $R_1, R_2, \ldots, R_m$. Each node R_i stores a table with two columns, one with values from column i of R and one with the primary key values from R, denoted as R_{id}.

Similarly, S is vertically partitioned into n data sites and T is vertically partitioned into k data sites.

Let Rd, Se, Ra, Sa, Sb, Tb denote the tables on the data sites which store $R.d, S.e, R.a, S.a, S.b, T.b$ columns, respectively.

The query above can be processed as a pipeline of operators as illustrated in Figure 4(b):

1) The select scan is processed at Rd and Se, which can be expressed as $\pi_{Rid}(\sigma_{R.d} > \alpha)$ and $\pi_{Sid}(\sigma_{S.e<\beta})$. This will return the qualifying Rids and Sids, which are then sent to Ra and Sa respectively. We assume the maximum cost for performing this step is on Rd.

- Scan the partition: I/O cost = $\frac{Rd}{B} \times t_{IO}$
- Iterate through blocks, iterate through tuples in each block to check the select predicate
  CPU cost = $\frac{Rd}{B} \times t_{Blk} + Rd \times t_{tup} + t_{func}$
- Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used):
  I/O cost = $2 \times \frac{(ssl_R \times |Rd| \times size_{Rid})}{B} \times t_{IO}$

- send resulting Rids to the storage node (if OTPM is used): NW cost = $\frac{(ssl_R \times |Rd| \times size_{Rid})}{P} \times t_p$
- send resulting Rids to Ra: NW cost = $\frac{(ssl_R \times |Rd| \times size_{Rid})}{P} \times t_p$

2) The Rids and Sids are sent to Ra and Sa respectively to access the corresponding σ_R records in Ra and σ_R records in Sa involving in the first hash join $Ra \bowtie_{R.a=S.a} Sa$. These records are hashed on the value of column a and sent to u query sites A_1, A_2, ..., A_u, which correspond with u buckets of the hash function H_1.
   - Scan the partition: I/O cost = $B(Ra)timest_{IO}$ where $\frac{ssl_R \times Ra}{B} \leq B(Ra) \leq \frac{Ra}{B}$
   - Iterate through blocks, iterate through tuples in each block and apply the hash function to each satisfying tuple: CPU cost = $B(Ra) \times t_{Blk} + ssl_R \times Ra \times t_{tup} + ssl_R \times Ra \times t_{func}$
   - Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used): I/O cost = $2 \times \frac{(ssl_R \times Ra)}{B} \times t_{IO}$
   - Send resulting tuples to the storage node (if OTPM is used): NW cost = $\frac{(ssl_R \times Ra)}{P} \times t_p$
   - Send resulting tuples to query sites: NW cost = $\frac{(ssl_R \times Ra)}{P} \times t_p$

3) Let R_{A_j}, S_{A_j} be the set of records sent from Ra, Sa to $A_j (j = 1, 2, \ldots, u)$. Each query site A_j will perform the join $R_{A_j} \bowtie S_{A_j}$. The resulting Sids are sent back to Sb.
   - Write and read temporary tables: I/O cost = $\frac{R_{A_j} + S_{A_j}}{B} \times t_{IO} \times 2$
   - Iterate through blocks, iterate through tuples in each block to build the hash table (for the tuples in the smaller table) and probe for matches (for the tuples in the other table), apply the hash function to each matching tuple: CPU cost = $\frac{R_{A_j} + S_{A_j}}{B} \times t_{Blk} + (|R_{A_j}| + |S_{A_j}|) \times t_{tup} + t_{func} + jsl_{S_{A_j}} \times |S_{A_j}| \times t_{func}$
   - Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used): I/O cost = $2 \times \frac{(jsl_{S_{A_j}} \times |S_{A_j}| \times size_{Sid})}{B} \times t_{IO}$
   - send resulting Sids to the storage node (if OTPM is used): NW cost = $\frac{(jsl_{S_{A_j}} \times S_{A_j})}{P} \times t_p$
   - send resulting Sids to Sb: NW cost = $\frac{(jsl_{S_{A_j}} \times S_{A_j})}{P} \times t_p$

4) The resulting Sids of the first hash join, i.e. $j = \cup_{j=1}^{u} \pi_{Sid}(R_{A_j} \bowtie S_{A_j})$ are sent back to Sb to access the corresponding records in Sb involving in the second hash join $Sb \bowtie_{S.b=T.b} Tb$. These records are then hashed on the value of column b using the hash function H_2 and sent to v query sites $Q_1, Q_2, \ldots, Q_v$. Simultaneously, Tb is also hashed by H_2 and sent to those v query sites. The cost to perform this step on Sb includes:
   - Scan the partition: I/O cost = $B(Sb) \times t_{IO}$ where $Sb \times \frac{\Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}|)}{|Sb| times B} \leq B(Sb) \leq \frac{Sb}{B}$
   - Iterate through blocks, iterate through tuples in each block and apply the hash function to each satisfying tuple: CPU cost = $B(Sb) \times t_{Blk} + \Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}| \times t_{tup} + \Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}|) \times t_{func}$
   - Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used): I/O cost = $2 \times \frac{Sb \times \Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}|)}{|Sb| \times B} \times t_{IO}$
   - Send resulting tuples to the storage node (if OTPM is used): NW cost = $\frac{Sb \times \Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}|)}{|Sb| \times B} \times t_p$ Send resulting tuples to query sites: NW cost = $\frac{Sb \times \Sigma_1^u (jsl_{S_{A_j}} \times |S_{A_j}|)}{|Sb| \times B} \times t_p$
   - The cost to perform this step on Tb includes: Scan the partition: I/O cost = $\frac{Tb}{B} \times t_{IO}$
   - Iterate through blocks, iterate through tuples in each block and apply the hash function to each satisfying tuple: CPU cost = $\frac{Tb}{B} \times t_{Blk} + |Tb| \times t_{tup} + |Tb| \times t_{func}$
   - Send resulting tuples to query sites: NW cost = $\frac{Tb}{P} \times t_p$

5) Let S_{Q_i}, T_{Q_i} are the set of tuples sent from Sb, Tb to $Q_i (i = 1, 2, \ldots, v)$. Each query site Q_j will perform the join $S_{Q_i} \bowtie T_{Q_i}$. The resulting Tids are sent back to all the T_i sites.
   - Write and read temporary tables: I/O cost = $\frac{T_{Q_i} + SQ_i}{B} \times t_{IO} \times 2$
   - Iterate through blocks, iterate through tuples in each block to build the hash table (for the tuples in the smaller table) and probe for matches (for the tuples in the other table): CPU cost = $\frac{T_{Q_i} + S_{Q_i}}{B} \times t_{Blk} + (|T_{Q_i}| + |S_{Q_i}|) \times t_{tup} + t_{func}$
   - Extra I/O cost to write/ read the results to/ from local disk (if complete materialization is used): I/O cost = $2 \times \frac{(jsl_{S_{Q_i}} \times |T_{Q_i}| \times size_{Tid})}{B} \times t_{IO}$
   - Send resulting Sids to the storage node (if OTPM is used): NW cost = $\frac{(jsl_{S_{Q_i}} \times T_{Q_i})}{P} \times t_p$
   - Send resulting Sids to Tb: NW cost = $\frac{(jsl_{S_{Q_i}} \times T_{Q_i})}{P} \times t_p$

6) The resulting Tids of the second join, i.e. $i = \cup_1^v \pi_{Tid}(S_{Q_i} \bowtie T_{Q_i})$ are sent back to all the T_i sites to get the final results.
   - Scan the partition: I/O cost = $B(T_i) \times t_{IO}$ where $T_i \times \frac{\Sigma_1^v (jsl_{S_{Q_i}} \times |T_{Q_i}|)}{|T_i| \times B} \leq B(T_i) \leq \frac{T_i}{B}$
   - Iterate through blocks, iterate through tuples in each block : CPU cost = $B(T_i) \times t_{Blk} + \Sigma_1^v (jsl_{S_{Q_i}} \times |T_{Q_i}|) \times t_{tup}$
   - Send resulting tuples to a central node: NW cost = $T_i \times \frac{\Sigma_1^v (jsl_{S_{Q_i}} \times |T_{Q_i}|)}{|T_i| \times P} \times t_p$

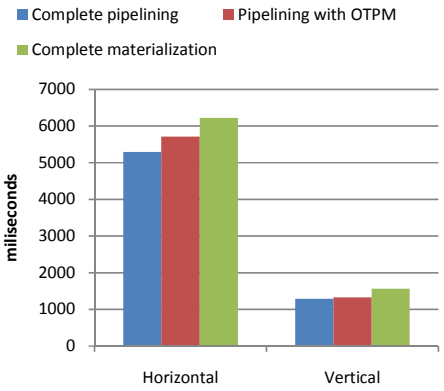


Fig. 5. Total running time without failure.

V. Experimental Evaluation

We create a simulator to evaluate the effectiveness and efficiency of our proposed fault-tolerance scheme. The input for the simulator includes the query plans and the costs as described in our analytical cost model. The simulator models different scheme of parallel processing and controls the query progress. We run the simulator for our example TPC-H query with SF=1. When using the OTPM scheme, we need nodes to store intermediate results. If the system has a fixed number of query sites, some of them have to operate as storage nodes when using the OTPM scheme. Thus, for a fair comparison, the number of query sites used in the OTPM scheme should be fewer than the number of query sites used in a complete pipelining (CP) scheme and a complete materialization (CM) scheme. This means that the amount of work at each query site in a system with the OTPM scheme is larger than that in a system with complete pipelining or complete materialization. Figure 5 shows the total running time to process the given query without failure. Complete materialization is the most expensive of the three schemes. Complete pipelining is obviously the cheapest. If the database is horizontally partitioned, the overhead cost of using CM compared to CP is 17.44% while that of OTPM is 7.78%. If the database is vertically partitioned, CM is 20.86% slower than CP while OTPM is 2.69% slower than CP.

A. Failure Handling at Select Site

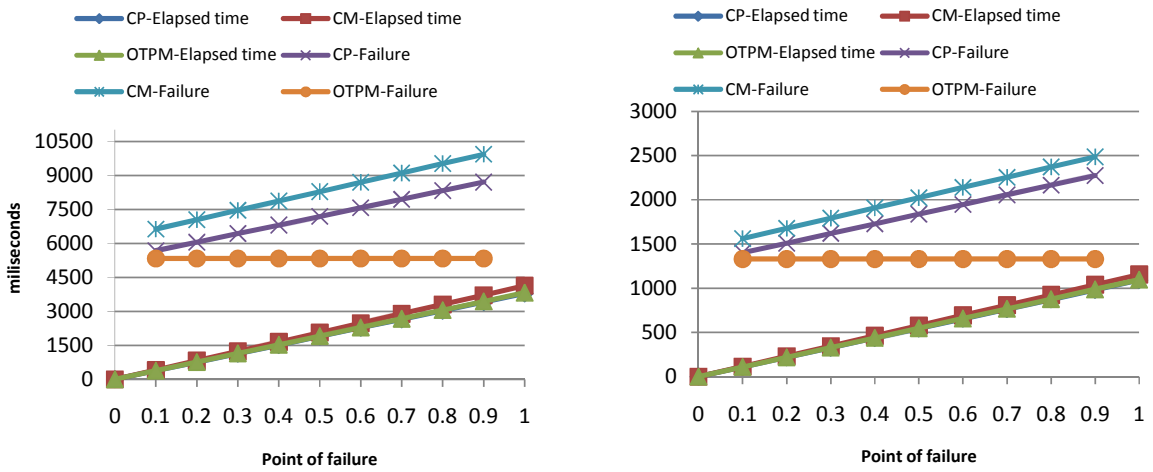
When there is a failure at a select site then the processing continues from the replicated data at the other sites. The replicated data is either chained or interleaved. If interleaved data replication is used, it will speed up processing after failure. Instead of scanning on one site, in case an operator node fails, the task is performed by $(m-1)$ operations concurrently on $(m-1)$ sites that store the interleaved replication of data. If chain replication is used, the failed scan will be performed at the site that stores the identical copy of the data as the failed node. Here we compute the cost for chain replication.

For complete pipelining, the query has to be restarted from the beginning. Thus, the total running time will be the original running time (without the event of failure) plus the time it takes to run from the beginning to the point of failure at the select site, i.e. the percent of the relation that has been scanned. In complete materialization, since all the intermediate results saved at the local site are lost when that site fails, the failed scan has to completely restart at the site which stores the data replica. Hence, the greater the percent of the relation has been scanned before failure, the longer it takes to finish the query. This is illustrated in Figure 6, in which the total running time of CP and CM (denoted as CP-Failure and CM-Failure) is proportional to the point of failure at the select site. CP-Elapsed and CM-Elapsed are the elapsed time for CP and CM respectively until the site fails.

For pipelining using our OTPM scheme, the alternate site can start scanning exactly where the failed scan left off. The scan at the alternate site will begin after the max $last\_pk$ tuple. We assume the cost to switch connections and to send checkpoint information from the operator trackers at downstream sites to the alternate site is negligible compared to the scan cost. So basically, the running time (OTPM-Failure) is nearly the same as original running time regardless of the point of failure, which makes this scheme the best of the three in failure handling, as shown in Figure 6.

B. Failure Handling at the First Join Site

Figure 7 shows the total cost, i.e. total running time, corresponding to three processing schemes when there is a single join site failure. The underlying join at each join site is a hash join. The smaller of the two relations is taken as the build relation, i.e. Part. For a horizontally partitioned database, we choose $u = 5$ when using CP and CM, $u = 4$ when using OTPM as the number of query sites to perform the parallel join. When using OTPM, at each join site A_i, there are 20000 tuples from Partsupp joining with 20000 tuples from Part, which are the results from the select scan. These fit in memory and can be cached locally at the select site. But for a larger database, e.g. table Partsupp with 205 million records and the query site with a memory of size 1Gb, the select results will exceed memory and must be materialized on a storage node. We assume that case, so upon A_i's failure, there are additional I/O costs and network costs for reading and sending tuples from the storage node to the alternate node. Here, we take the point of failure as the percent of tuples from the larger-size operand, i.e. the probe operand, which has been processed so far. The unprocessed part of the probe relation is resent while the build relation will have to be fetched entirely from the storage node. Thus, the recovery overhead of the OTPM scheme is inversely proportional to the point of failure. This makes the total cost of using OTPM decrease when failure happens at later points as evident from Figure 7. However, the join site in the OTPM scheme has to process more data than in the other two schemes since we use smaller number of query sites. Moreover, earlier failure requires a larger amount of data to be read from the storage node and sent to the alternative node as its input. That is why from Figure 7(a), we can see that complete materialization is more efficient than



(a) Horizontal (b) Vertical

Fig. 6. Relative performance of the schemes when the select site fails.

OTPM when the failure happens early (at points of failure ¡ 30%). However, this does not happen when the database is vertically partitioned (Figure 7(b)). Here we choose $u = 3$ for CP/CM and $u = 2$ for OTPM. The reason is that we only process on the join column so the size of input data decreases significantly compared to the horizontal partitioning approach. Thus, there is just a small difference between the sizes of data which the join sites in OTPM scheme and complete pipelining or complete materialization have to process. The difference of incurred recovery cost between two consecutive points of failure in each scheme is also small. This is illustrated in Figure 7(b) where the lines have slight slopes.

CP-Failure is the most expensive and is a linear function of the point of failure at a join site. It is because for complete pipelining, when a failure happens at a join site A_i, failure handling means completely restarting the query. All the work which has been done so far is lost. Therefore, it takes the sum of the elapsed time until the point of failure and the total running time without any failure. If complete materialization is used, the failed operation is restarted at an alternative node. Only the work from the beginning of step 2 (see section IV-A) to the point of failure is lost. Intermediate results saved at upstream sites are rescanned and sent to the alternate node as inputs for the join. The costs during recovery are the cost to supply the input for the join and the cost to reprocess the work from the beginning of step 2 to the point of failure. In our example, the selectivity of the select scan and the first join are quite high, i.e. the proportion of qualifying tuples is low (10% at the select scan and 20% at the first join) so that materializing intermediate results is more efficient than complete restarting when there is failure at A_i. Hence, CM-Failure lies below CP-Failure in Figure 7. With a lower selectivity query, the cost for complete materialization scheme might dominate.

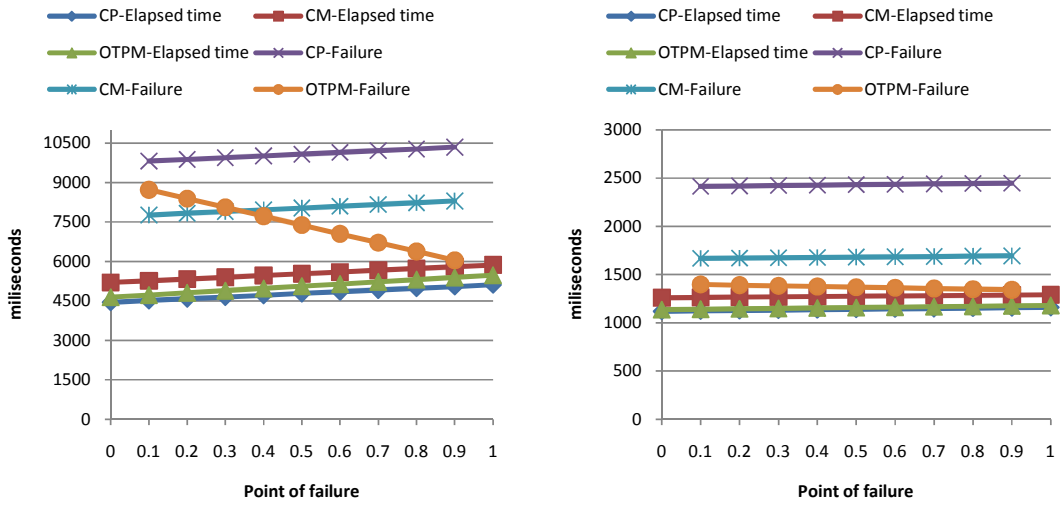
C. Failure Handling at the Second Join Site

The number of records generated after the first join is 9624 which are assumed to be equally hashed between the different nodes of the second join. Also from the third relation (Supplier), the records are hashed to the join sites. The same formula as the previous join would apply here. At the second join site Q_i, the size of input data is just about 12.5% of that at the first join site A_i. Hence, the relative performance of the three schemes is similar to the previous Figure 7(b). As shown in Figure 8, pipelining with OTPM still outperforms the other two in query processing with the event of failure.

VI. Related Work

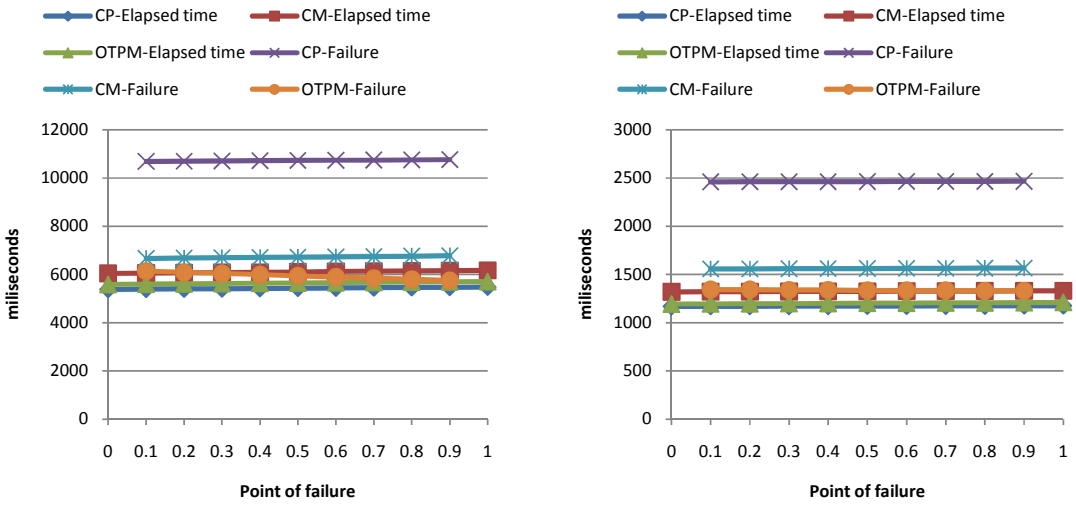
Several methods [12, 13] have been presented to support query restarting in centralized query processing. In [12], each query operator performs checkpointing asynchronously at times when its state is minimal. Checkpointing or dumping the state of the whole query, whichever incurs smaller restarting cost, is chosen to perform suspending and resuming the query. A different approach [13] is to cache some immediate results during execution so that they can be skipped during restarting the whole query. These methods are applied for handling intended query stopping or suspending while we aim to handle unintended query stopping due to site failure.

In data stream management systems (DSMSs), fault tolerance has been well studied to support stream based applications. These applications have to process large quantities of data continuously pushed into the systems, so that these systems must keep processing even when failures occur. Fault tolerance techniques in DSMSs usually employ replication of computation. The approaches in [14, 15] require a backup server for every primary server, running the same computation in parallel with the primary server, which can provide fast recovery when a server fails. However, these approaches incur significant runtime overhead and sometimes may not take the advantage of the backup server since the failure probability of the backup server is the same as of the primary server. Hwang *et. al.* [16] presented the concept of *passive standby* server which stores the state of the primary server periodically. If a server goes down, its standby server recovers the



(a) Horizontal (b) Vertical

Fig. 7. Relative performance of the schemes when the first join site fails.



(a) Horizontal (b) Vertical

Fig. 8. Relative performance of the schemes when the second join site fails.

failed operation from the last checkpoint. Another approach presented in [16] is *upstream backup* where the primary server logs its produced data in an output queue. When the primary server fails, the backup server has to reprocess all the data output logged at upstream server. However, the paper does not describe how to log the output data, whether in memory or on disk. Both passive standby and upstream backup cause re-sending and re-processing some data that has already been processed and sent downstream, which results in duplicate data at the final output if failure occurs. In contrast, our approach restarts the failed operation at the point of failure and gives the exact final output as in the original process.

Unlike stream based applications which run continuously to process a dynamic sequence of data input, data analytical applications rely on a set of static data sources. Therefore, mechanisms in parallel query processing should be less resource consuming and provide fast runtime. To the best of our knowledge, there has not been much work on fault tolerance in parallel query processing. Some clustered DBMS implementations [17, 18] also use the passive standby scheme. Jim Smith and Paul Watson [19] modified OGSA-DQP [20], a publicly distributed query processing system for the grid, to support fault tolerance using a rollback recovery protocol presented in [21]. Each operator node saves its output tuples in a recovery log and inserts a checkpoint marker to each block of tuples sent downstream. After traveling through a given number of nodes, the checkpoint marker will be sent back to the node where it was created so that all tuples preceding this checkpoint in the recovery log can be truncated. During recovery, the recovery log at an upstream node is sent to the replaced node to restart the failed operation. Duplicate tuples produced by the new node will be discarded at downstream nodes. An approach

close to our own was proposed by Hauglid *et. al.* [22] in which each operator node inserts a number to each tuple packet sent downstream. This number is the ID of the last processed tuple in order to output the tuple packet or the number of tuples which have been sent downstream, depending on whether the operator is stateless or stateful respectively. The replace node does the work of the failed node from the beginning and starts sending out results when it reaches that number. This approach prevents sending duplicate tuples from the new node but does not prevent reprocessing tuples during failure recovery. Our approach prevents both resending and reprocessing tuples by tracking operator progress in more details, provides substantial savings in both network cost and CPU cost during recovery.

VII. Conclusion

We have introduced OTPM, an efficient fault-tolerance scheme for parallel query processing of data analytical workloads. Our scheme benefits read-intensive queries as in most analytical workloads. There is an operator tracker at each node which stores a small information about the progress running at its upstream nodes. In addition, intermediate tuples are partially materialized. The OTPM scheme guarantees that if a node fails, its task can be resumed at an alternate node thus the elapsed progress is not wasted. We study the effectiveness of our approach under different system settings through an analytical cost model and a set of simulation-based experiments. The analytical and experimental results show that our approach incurs only a small overhead during execution and outperforms the two widely used schemes in parallel DBMSs and MapReduce-based systems at failure handling.

Acknowledgment

This work is partially sponsored by grants from NSF CISE NetSE and CrossCutting program, an IBM SUR grant, an IBM faculty award and Intel ISTC.

References

[1] Dbms2's blog. [Online]. Available: http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive/

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, 2009.

[4] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. SIGMOD'09*, 2009, pp. 165–178.

[5] D. J. Dewitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, pp. 85–98, 1992.

[6] S. Madden, D. J. DeWitt, and M. Stonebraker. Database column blog. [Online]. Available: www.databasecolumn.com/2007/10/database-parallelism-choices.html

[7] M. T. Özsu and P. Valduriez, "Distributed and parallel database systems," *ACM Computing Surveys*, vol. 28, pp. 125–128, 1996.

[8] M. Mehta and D. J. DeWitt, "Data placement in shared-nothing parallel database systems," *The VLDB Journal*, vol. 6, no. 1, pp. 53–72, 1997.

[9] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, "Toward a progress indicator for database queries," in *Proc. SIGMOD'04*, 2004, pp. 791–802.

[10] The TPC-H benchmark. [Online]. Available: http://www.tpc.org/tpch/

[11] D. J. Dewitt, S. R. Madden, D. J. Abadi, D. J. Abadi, D. S. Myers, and D. S. Myers, "Materialization strategies in a column-oriented DBMS," in *Proc. ICDE'07*, 2007, pp. 466–475.

[12] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang, "Query suspend and resume," in *Proc. SIGMOD'07*, 2007, pp. 557–568.

[13] S. Chaudhuri, R. Kaushik, A. Pol, and R. Ramamurthy, "Stop-and-restart style execution for long running decision support queries," in *Proc. VLDB '07*, 2007, pp. 735–745.

[14] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proc. SIGMOD'05*, 2005, pp. 13–24.

[15] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. SIGMOD'04*, 2004, pp. 827–838.

[16] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. ICDE'05*, 2005, pp. 779–790.

[17] IBM, "High availability and disaster recovery options for DB2 on Linux, Unix, and Windows," IBM Redbooks, 2009.

[18] Oracle, "Mysql cluster 7.0 & 7.1: Architecture and new features," A MySQL technical white paper, 2010.

[19] J. Smith and P. Watson, "Fault-tolerance in distributed query processing," in *Proc. IDEAS'05*, 2005, pp. 329–338.

[20] M. N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, P. Watson, A. A. Fernandes, and D. J. Fitzgerald, "OGSA-DQP: A service for distributed querying on the grid," in *Proc. EDBT'04*, 2004, vol. 2992, pp. 858–861.

[21] J. Smith and P. Watson, "A rollback-recovery protocol for wide area pipelined data flow computations," Tech. Rep., 2004.

[22] J. O. Hauglid and K. Nørvåg, "PROQID: partial restarts of queries in distributed databases," in *Proc. CIKM'08*, 2008, pp. 1251–1260.