Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database

Christopher Yang [1], Christine Yen [2], Ceryen Tan [3], Samuel R. Madden [4]

CSAIL, MIT
77 Massachusetts Ave, Cambridge, MA 02139, USA
[1]yangc@mit.edu
[2]cyyen@mit.edu
[3]ctan@mit.edu
[4]madden@csail.mit.edu

***Abstract*— In this paper, we describe a scheme for tolerating and recovering from mid-query faults in a distributed shared nothing database. Rather than aborting and restarting queries, our system, *Osprey*, divides running queries into *subqueries*, and replicates data such that each subquery can be rerun on a different node if the node initially responsible fails or returns too slowly. Our approach is inspired by the fault tolerance properties of MapReduce, in which map or reduce jobs are greedily assigned to workers, and failed jobs are rerun on other workers.**

Osprey is implemented using a middleware approach, with only a small amount of custom code to handle cluster coordination. Each node in the system is a discrete database system running on a separate machine. Data, in the form of tables, is partitioned amongst database nodes and each partition is replicated on several nodes, using a technique called *chained declustering* [1]. A coordinator machine acts as a standard SQL interface to users; it transforms an input SQL query into a set of subqueries that are then executed on the nodes. Each subquery represents only a small fraction of the total execution of the query; worker nodes are assigned a new subquery as they finish their current one. In this greedy-approach, the amount of work lost due to node failure is small (at most one subquery's work), and the system is automatically load balanced, because slow nodes will be assigned fewer subqueries.

We demonstrate Osprey's viability as a distributed system for a small data warehouse data set and workload. Our experiments show that the overhead introduced by the middleware is small compared to the workload, and that the system shows promising load balancing and fault tolerance properties.

I. Introduction

Most existing distributed database systems handle node failures during query execution by aborting the query and (possibly) restarting it. This is a perfectly reasonable approach for very short OLTP-style queries, but for longer running analytical (OLAP) warehouse queries that may run for minutes or hours, running on large clusters of machines where faults may be common, it is highly preferable to be able to avoid re-running queries from the beginning when a node fails.

To illustrate the problem, consider a shared-nothing distributed data warehouse that runs a series of 6 reports every day. Suppose each report takes 3 hours to generate on a 100 node database with 800 disks (anecdotally, 100 nodes are about the size of the largest Teradata clusters[1]; vendors like Netezza sell clusters with as many as 1800 nodes). Suppose that the OS on a single node crashes once every 30 days (once every 8×30 reports, since each report runs for 1/8th of day), and that a single disk fails once every 2 years (once every $8 \times 365 \times 2$ reports). Then, the probability that no node fails while a query runs is $(1 - \frac{1}{8 \times 30})^{100} = 65\%$, meaning that there is a 35% chance that a crash will occur during a given report query, wasting 1.5 hours of work on average on each crash. Such wasted work likely means that some reports will not be generated that day, or that the warehouse won't be able to load all of that day's data during the six hours of downtime while not generating reports. Similarly, there is a $1 - (1 - \frac{1}{8 \times 365 \times 2})^{800} = 13\%$ chance that a disk fails while a query is running. While real systems may have different numbers of disks or CPUs, and failure rates may vary somewhat (in particular, disk storage system vendors go to great lengths to hide the failures of individual drives), it is clear that as database clusters get larger, a way to restart queries mid-flight will be important in order for warehouses to handle their daily loads.

To address this problem, we have built a distributed shared-nothing database system, called *Osprey*, that provides the ability to detect and to recover from failures (or slow nodes) in long-running queries. Osprey uses a middleware approach, where a central coordinator dispatches work-units (in the form of query fragments) to workers. Each worker is simply an unmodified DBMS (in this case, Postgres). Data is partitioned amongst workers, and each partition is replicated across several nodes, using a technique called *chained declustering* [1], which limits data unavailability due to machine failures. The coordinator runs our custom middleware code while presenting a standard SQL interface to users. Users input a SQL string, which the coordinator transforms into a set of subqueries that the workers execute. Typically, a single input query is broken into hundreds or thousands of subqueries, depending on how the tables of the database are partitioned. The system is naturally load balanced and fault tolerant, because slow (or

[1]http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/

dead) machines will receive fewer subqueries to execute while fast machines will do relatively more. We explore a range of techniques to schedule subqueries amongst workers, including techniques to reassign slow workers' jobs to faster workers so the query finishes as quickly as possible.

Our approach is loosely inspired by the approach taken by MapReduce [2], where map and reduce jobs are run on a set of worker nodes by a scheduler. When one worker fails (or slows), its jobs are rescheduled on another worker, allowing that task to complete without restarting. Unlike MapReduce, Osprey does not rely on a distributed file system like GFS, but uses chained declustering to ensure data is available on one or more replicas. Also, unlike MapReduce, our system runs general SQL queries rather than simple MapReduce jobs.

In summary, the major contributions of this work are as follows:

- We describe how to decompose data warehouse queries into sub-queries of relatively large size that can be executed by our middleware-based job scheduler, enabling mid-query restartability and adaptation to slow nodes.
- We show that chained-declustering is an effective technique for replicating data in this setting.
- We investigate several different job-scheduling techniques for jobs in this context.
- We show that the overall approach taken in Osprey is able to achieve linear-speedups (a factor of 9.7× on an 8 node cluster on the SSB [3] data warehousing benchmark) on a parallel cluster while simultaneously adapting to slow or overloaded nodes.

The remaining of the paper is organized as follows: Section II describes related work. Section III provides a high-level view of the design of the system, as well as our specific strategies for job scheduling and load balancing. In Section IV, we present results, which demonstrate the viability of our middleware approach and also promising load balancing and fault tolerance properties of the system. Finally, we discuss the implications of our system and possible future work in Section V.

II. Related Work

Osprey is related to several existing projects, which we describe here.

A. MapReduce

MapReduce is Google's distributed solution for web-scale data processing [2]. MapReduce automatically parallelizes programs to run on a cluster of commodity hardware. This automation comes at the cost of the expressiveness of the input program. Programs are written as *map* functions, reading in an individual item in the original data set and outputting an intermediate tuple, and *reduce* functions, which merges the intermediate tuples created by the *map* step into the final output. All *map* operations can be performed in parallel, and all *reduce* steps run in parallel (after the *map* phase has completed).

Because it is designed to be run on a cluster of commodity hardware (where disparities in processing power and availability are potentially drastic), MapReduce employs two strategies for load balancing and fault tolerance. (1) Worker nodes are assigned *map* and *reduce* tasks as quickly as they finish them, in a so-called "greedy" fashion. This leads to natural dynamic load balancing properties, as slow machines will be assigned less work as they complete tasks more slowly. Fast machines, in contrast, will be assigned more work as they finish their tasks relatively quickly. (2) Minimize the effect of "straggler" machines (degenerately slow workers) by re-executing tasks, which was shown to drastically shorten the total execution time of a job [2]. MapReduce uses the Google File System (GFS) to distribute the original data and intermediate results amongst the cluster machines.

Osprey is similar to MapReduce in that they are both designed to run on clusters of heterogeneous hardware performance. It adapts the MapReduce strategy of parallelization by breaking up a program (or SQL query) into smaller, parallelizable subqueries. Osprey also adapts the load balancing strategy of greedy assignment of work.

However, Osprey differs fundamentally from MapReduce in that Osprey is designed as a SQL system from the ground up. While there are some similarities between the MapReduce programming model and the limitations Osprey places on SQL queries, we retain the declarative style of SQL. Osprey's middleware approach means that we get low-level SQL query optimizations for free – clustering of tables, indexes on commonly used fields, statistics on data distribution, and integrity constraints between tables can take advantage of patterns in the data that an imperative style like MapReduce would be hard-pressed to duplicate manually.

B. MapReduce in Databases

Two commercial database systems have emerged that retrofit MapReduce functionality into existing database systems. Greenplum transforms MapReduce code into a query plan that its proprietary distributed SQL engine can execute on existing SQL tables [4]. Aster implements something similar, where MapReduce functions can be loaded into the database and invoked from standard SQL queries in Aster's distributed engine [5]. To the best of our knowledge, neither of these systems implements MapReduce-style fault recovery.

HadoopDB shares our middleware approach, similarly using PostgreSQL servers in their database layer, but uses Hive and Hadoop for query transformation, job scheduling and replication [6]; Osprey uses chained-declustering replication and our own scheduling methods. BOOM [7] attempts to graft a declarative language over a MR framework, but does not use a middleware approach or even use SQL.

C. Middleware

Kemme, Patiño-Martínez, and Jiménez-Peris have done much work in data replication in middleware approaches [8], [9], [10], and have focused on efficient ways to keep replicas

synchronized. Their approaches to fault tolerance are fundamentally different from Osprey. In [9], replicated standbys are brought online as machines fail, which is very different from Osprey's dynamic approach to fault tolerance. Furthermore, our approach allows long-running queries to finish, despite individual worker failure. In a replica-based recovery scheme, queries would have to be re-executed in the case of failure [9].

Osprey does share the idea of creating a client-transparent database using a middleware approach, with Pronto [11] and Lin et al. [9].

D. Chained Declustering and Dynamic Load Balancing

Osprey replicates data using a technique called *chained declustering* [1], which was created by Hsiao and DeWitt for use in Gamma, a distributed shared-disk database [12]. Chained declustering provided better data availability than other strategies (such as RAID and Teradata's replication strategy [1]), and offered a static load balancing strategy for machine failure.

Golubchik et al. [13] adapted the chained declustering load balancing to a dynamic setting, presenting two scheduling algorithms that Osprey adapts. Their analysis of their scheduling algorithms was in the context of scheduling blocks to read from shared disk controllers, and the performance results of their schedulers was only a simulation. A more thorough discussion of chained declustering and Golubchik et al.'s dynamic schedulers is presented in Section III-D.1.

III. Osprey

Osprey's approach to fault-tolerant distributed query execution employs three key ideas:

- Its middleware approach allows us to cleanly separate the low-level components (the actual execution of SQL queries) from higher level fault-tolerance concerns;
- Fault tolerance is enabled by replicating data using chained declustering, and partitioning each query into a number of subqueries, each of which can be executed independently on one or more replicas. A coordinator node keeps track of which chunks are still outstanding and allocates chunks to workers.
- Load balancing is achieved via a dynamic scheduler. Osprey implements three subquery schedulers, whose performance we compare in Section IV.

Before describing the system in detail, we note a few assumptions that we made in building Osprey:

- *Data warehouse workload.* Osprey is designed for data warehouse applications. We have assumed that the tables are arranged in a star schema, with any number of dimension tables and a single fact table, typically many orders of magnitude larger than any of the dimension tables. We make this assumption because we feel that warehouse systems are the most likely to benefit from the types of fault tolerance techniques we discuss; transactional systems are unlikely to need mid-query fault tolerance, as queries are typically very short. We note that this workload is what the initial versions of the data warehousing products from many commercial data warehouse vendors (e.g., Netezza and Vertica) supported; Vertica, for example, did not support non-star schema joins until version 2 of their database was released, and was still successful in making large sales to many customers. We briefly discuss how we might extend Osprey to handle more general long-running query workloads (for example, with multiple large tables) in Section III-G below.
- *Read-only, long-running deterministic query workload.* The scheduling schemes presented here assume that queries run for long enough to justify the overhead of a load balancing strategy. We also assume that there are few on-line updates, with the bulk of new rows coming into the database via large batch updates, which is consistent with a warehouse workload. Finally, we recognize that non-deterministic queries would not behave correctly in Osprey, and that our assumed data warehouse workload contains only deterministic queries.
- *Heterogeneous load on cluster machines.* We assume that machines in the system do not all perform identically. This is consistent with Google's commodity hardware approach to distributed systems [2], but is also consistent with many other real world possibilities: machines of different age, machines that are loaded by other queries or non-database work, etc. We show that Osprey performs quite well in such an environment.

In the remainder of this section, we present the details of Osprey's design, in five parts. First, we give a brief overview of the architecture. Next, we discuss the data model, and then the query workload that we the system is optimized for, third. Fourth, we discuss our scheduling and backup schemes. Finally, we end with a short sample query execution walkthrough, and a discussion of how to generalize Osprey to more complex query workloads.

A. Architecture

An Osprey cluster is composed of a *coordinator* and n *workers*. The coordinator presents a SQL interface to users (queries that Osprey cannot execute are rejected – see Section III-C.1). The coordinator runs the Osprey middleware – workers are off-the-shelf, unmodified database servers (Postgres, in our case). Only the coordinator runs custom code. Workers and coordinator are assumed to be physically distinct systems. Figure 1 summarizes the cluster layout.

The basic operation of the system is as follows: When the coordinator is asked to compute a query, it spawns a new *query manager* thread. The query manager is responsible for coordinating the workers to finish a given query. It implements a scheduler protocol, handing out chunks to process to each worker thread. The individual job is responsible for translating the SQL it was given by the coordinator into SQL appropriate to run directly on the worker. The query manager also marshals the intermediate results from each chunk from each worker, merging the results (and doing any appropriate aggregation processing) for the coordinator to return to the user.

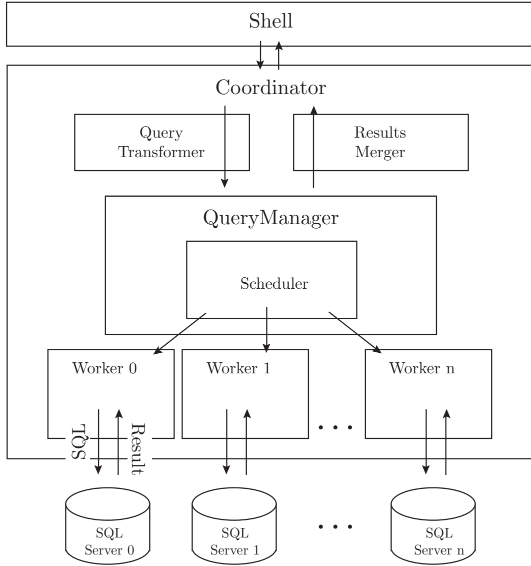

Fig. 1. Osprey architecture overview. An Osprey system consists of a coordinator which transforms and manages queries, manages a set of discrete Postgres SQL servers on worker machines, and returns the merged results.

W1	W2	W3	W4
A_1	**B_1**	**C_1**	**D_1**
A_2	**B_2**	**C_2**	**D_2**
B_1	C_1	D_1	A_1
B_2	C_2	D_2	A_2

Fig. 2. Using chained declustering to replicate the partitions A, B, C, D with a backup factor of $k = 1$. Data is unavailable only if $k+1$ adjacent machines simultaneously fail. Here we show each partition with $m = 2$ chunks each. Bold indicates partitions for which a worker is the primary.

There are n worker threads running on the coordinator machine, corresponding to n external database servers. Each worker is allowed to run an arbitrary number of queries at a time, although in practice, we limited this number to be small.

B. Data Model

As stated above, Osprey assumes a star schema for the database tables, where a fact table is related to several dimension tables. The fact table is assumed to be much larger than the dimension tables combined.

For a cluster of n worker machines, we use a hash partitioning scheme to divide the fact table into n partitions. These n partitions are replicated k times (for a total of $k + 1$ copies) onto k worker machines using *chained declustering* [1]. Chained declustering works as follows: Partition i is stored on Worker i. The k backups of Partition i are stored on workers $i + 1, \ldots, i + k$ (modulo the number of machines n). Figure 2 shows how chained declustering backs up the partitions $A, B, C,$ and D with a backup factor of $k = 1$.

The partition and its backup form a "chain" in the system – the partition is only unavailable if workers $i, \ldots, i + k$ all simultaneously fail. If a worker fails with probability p and if the failure of a worker in the system is an independent event, then the probability that a *particular* $k+1$ machines simultaneously fail (and thus a particular partition is unavailable) is p^{k+1}. This analysis also suggests machines adjacent in the "chain" should not share external resources, such as power source or network switch, lest a single failure in a common resource "break" the chain. Google's GFS takes similar precautions, preferring to keep backups of data off the same power grid or network as the primary [14].

Each partition is further segmented into m *chunks* in a round-robin fashion. Chunks are stored physically as tables in SQL server tables on worker machines. Each is identified by a globally unique table name, comprised of the base table name, parent partition, and chunk number. For example, Chunk 1 of Partition 2 of the `lineorder` table is named `lineitem_2_1`.

Dimension tables are replicated in full on each worker machine's database, as they are assumed to be small relative to the size of the fact table (Section III-G discusses how we can relax this limitation.)

C. Query Model

Partitioning the fact table into *chunks* lets us parallelize a query that runs over the entire fact table. With only slight modification to the original user query, we can compute the result of that original query by substituting the chunk for the fact table, and combining the results of all chunk subqueries. We describe how Osprey modifies user queries and assigns the execution of those subqueries here.

1) Query Transformation and Result Merging: Osprey transforms a user query into a set of *subqueries* that can be run in parallel on each of the workers. Because the fact table is partitioned across machines, Osprey can only execute queries in which the fact table appears exactly once in the `FROM` clause. There are no restrictions on the type of aggregation or filtering used – merely that self-joins are disallowed. Dimension-only queries are trivially executed in Osprey – the coordinator randomly chooses a worker to execute the unmodified query and returns the results.

A prototypical query might look like this:

```
select partnum, partname
from facttable, dim1, dim2, dim3
<...join conditions...>
where partnum between 0 and 10
```

Notice that the result of running this query could be computed as the union of the results of running queries of the form

```
select partnum, partname
from facttable_chunkX, dim1, dim2, dim3
where partnum between 0 and 10
```

where `facttable_chunkX` represents the set of all horizontally-partitioned *chunks* of the fact table. Chunks are physically stored on workers as full SQL tables, so the query transformer simply replaces the name of the fact table with

the chunk table name to create the subqueries. This set of queries can be run in parallel without affecting correctness of the result.

Osprey's *query transformer* breaks up a user's query into these smaller *subqueries*; the *results merger* combines the results of the subqueries to form the final query result. These steps are trivial for user queries that are simple filters and selects, but queries that involve grouping and aggregation require some care.

Consider a query that involves aggregation:

```
select partnum, count(partnum), min(price)
from facttable, dim1, dim2, dim3
<...where conditions...>
group by partnum
```

The query transformer can pass through this original query and substitute `facttable` for the names of the chunks. The results of these individual subqueries may look like this:

chunk	partnum	count(partnum)	min(price)
1	1006	20	425
2	1006	54	318

As in distributed databases that perform partial preaggregation [15], another aggregation is necessary to compute the final aggregate. In particular, the results merger must (1) group on `partnum`, (2) sum the subquery field `count(part-num)` to obtain the true count for part number 1006, and (3) take the minimum of `partprice` to find the true minimum for the part number. Aggregates like counting, minimum/maximums, and summing, can be dealt without requiring additional data; Gray et al. termed these aggregates as *distributive* functions [16].

Averaging over a column, say:

```
select partnum, avg(partprice)
from facttable, dim1, dim2, dim3
group by partnum
```

requires the query transformer to modify the original query slightly. We clearly cannot average the averages from each subquery directly, but we can compute the true average by having subqueries return the sum and count of the averaged column:

```
select partnum, sum(partprice),
       count(partprice)
from facttable_chunkX, dim1, dim2, dim3
group by partnum
```

The result merger simply computes the average of `partprice` by summing over `sum(partprice)` and `count(partprice)` from each subquery and dividing. Aggregating functions like averaging are *algebraic* functions [16], and can all be dealt with in a similar fashion.

Osprey's behavior can be summarized as follows: (1) subqueries are generated by replacing the fact table name with the name of the chunk, (2) `GROUP BY` fields and `WHERE` clauses are passed through unmodified, (3) average aggregates are replaces with sums and counts, and (4) result from each subquery are combined by the results merger, either as a union of result rows (if no aggregation is present) or as an aggregate the result rows (as described above).

Clearly, the above decomposition process doesn't support all types of queries (e.g., complex nested queries), but it is compatible with a large class of useful queries. Some more complex expressions – like nested queries or self-joins – might require redistribution of data mid-flight, which we do not currently support (see Section III-G for a discussion of how to add support for such queries.) MapReduce has similar limitations (it can redistribute data mid-flight, but requires the execution of another MR job); the restrictions we place on queries provide a programming model with similar expressiveness to its *map* and *reduce* functions. However, Osprey retains all of the advantages of DBMS's over MapReduce (e.g., indexing, schema management, crash recovery).

2) Update Queries and Transactional Processing: Osprey, as it is currently implemented, deals with read-only queries. As it does not deal with updates, our implementation omits any transactional machinery entirely. We felt this was an acceptable limitation as data warehouses traditionally bulk load data in a process usually requiring a temporary halting of query capabilities – already implemented in Osprey. Providing support for online updates, insertions, and deletions is not theoretically difficult but is not implemented because this capability was not needed for the benchmarking. We propose a possible implementation of write capability here.

The basic idea for update and deletion queries is to transform them into subqueries (using the same approach as for read-only queries) and issue them against each of the chunks on each of the database server instances on the worker nodes. To ensure that such updates are done transactionally, two-phase commit is needed. Postgres does support external two-phase commit, which should make this feasible.

For inserts, a single chunk of a single node is selected at random and the results are added to that chunk. If the database grows substantially, it may be necessary to divide some chunks into smaller ones.

Finally, to prevent running queries from seeing new data added during query execution, it would be necessary to run read-only queries over many chunks as a part of a two-phase commit transaction. Providing this level of isolation may not be necessary in many warehousing settings where large analytical queries are unlikely to be affected by one or two additional (or missing) results.

D. Subquery Execution Scheduling

While the chunks can be executed in parallel, the number of chunks (and thus subqueries) far exceeds the number of workers, so Osprey must schedule the execution of those subqueries to minimize the total runtime of the query, in the face of dynamically changing loads on the worker machines. Job scheduling in distributed systems is a well-studied problem; an overview of scheduling in general is beyond the scope of this paper, but we have implemented three straightforward scheduling algorithms in Osprey that we describe here.

1) Load Balancing Through Greedy Workers: Partitions (and thus chunks) and their backups are distributed amongst the workers using chained declustering. Hsiao and DeWitt

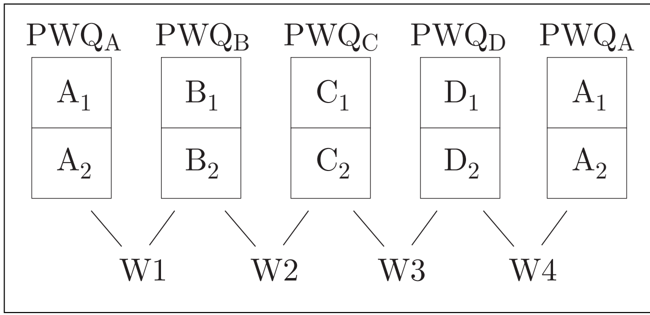

Fig. 3. Each worker can execute subqueries from one of two partitions. A_i represents the ith chunk of Partition A. Here we show a cluster of $n = 4$ workers with a backup factor of $k = 1$. Osprey uses a variety of scheduling algorithms to assign PWQ subqueries to workers.

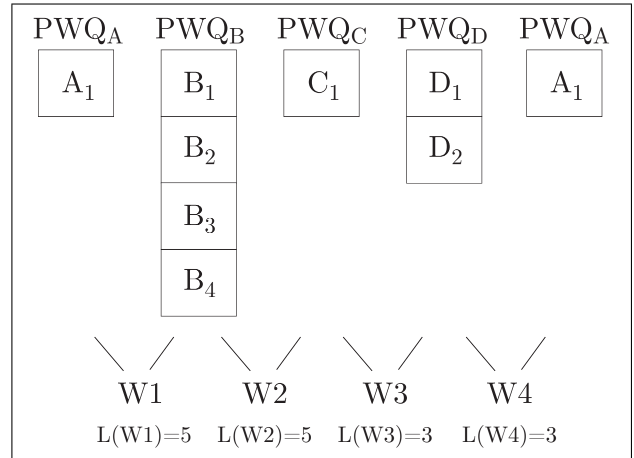

Fig. 4. A constructed situation with worker W3 about to be scheduled. L(W) represents the number of jobs available to worker W. LQF would assign job D_2 to W3, as D is the longer of the two queues. Majorization, however, would assign W3 C_1 in order to lighten the load on W2 from five remaining jobs to four. Golubchik showed that majorization could provide a 19% speedup over LQF.

recognized the use of chained declustering for load balancing under disk loss [1]; Golubchik et al. showed how chained declustering could be used for dynamic load balancing of shared disk controllers [13], and demonstrated their results through several simulations.

The basic idea is that a slow-down (or failure) of one disk can be compensated by its neighbors in the "chain." Those neighbors, in turn, are compensated for by *their* neighbors – thus, the load is balanced across all workers. We have adapted chained declustering and Golubchik's schedulers to work within Osprey's architecture.

In general, not every worker necessarily has access to all partitions – so each can only execute subqueries on chunks of partitions that it is storing locally. Osprey groups subqueries of chunks by partition, forming *partition work queues* (PWQs). For n workers, there are n partitions and thus n PWQs. Figure 3 shows the PWQs for a cluster of $n = 4$ machines, with a backup factor of $k = 1$. In this figure, each worker thread has two PWQs available from which it can pull subqueries. PWQs for each partition are maintained by the coordinator.

Workers are assigned a new subquery by the coordinator as soon as they finish processing their current query – we call this the *greedy worker* approach. Dynamic balancing arises naturally using such an approach because slow (or dead) workers will execute their subqueries more slowly and will thus execute fewer subqueries overall.

2) Subquery Scheduling: The PWQ that workers are assigned tasks from is chosen by a scheduling algorithm. We have implemented three:

1) *Random.* A PWQ is randomly chosen.
2) *Longest Queue First (LQF).* The PWQ with the most unexecuted subqueries is chosen. This is a good intuitive heuristic for minimizing the overall time to finish the execution, although [13] showed that LQF is not optimal.
3) *Majorization.* A faster scheduling algorithm uses the idea of vector *majorization* – we interpret the number of subqueries left to be executed in the PWQs as a vector. Majorization essentially tries to minimize the difference in PWQ lengths – occasionally choosing a job from the shorter PWQ "for the greater good," or to help another worker that may be further behind. Golubchik et al. describes majorization in greater detail and shows that it outperforms LQF by as much as 19% in a simulation study [13]. Figure 4 shows an example.

As subqueries are assigned to workers, the coordinator marks them as *assigned* and removes them from the queue. When the worker completes the subquery, it returns the result to the coordinator, who marks the subquery as complete. If the PWQs a worker can be assigned work from are all empty, the worker will re-execute an already-assigned subquery that another worker with which it shares a PWQ has not finished. Like MapReduce, Osprey uses re-execution to minimize the effect of "straggler" workers (that are executing much slower) and failed workers [2].

E. Execution Overview

In this section, we briefly summarize the execution of a query in Osprey. Figure 5 shows this in diagram form.

1) Osprey receives a query from the user. The query transformer converts this query into a set of subqueries. Partition work queues are created, one per partition. The subqueries are added to these PWQs.
2) While the PWQs are not empty, we attempt to schedule a subquery for each worker:
   a) The query manager cycles through all workers, looking for one not already running a subquery (*available*). Workers are pinged to see if they are still up – if a worker fails to respond to the ping, it is marked as *down* and will not have subqueries assigned to it until it responds to a subsequent ping.
   b) The scheduling algorithm assigns workers an outstanding subquery to execute. The subquery is marked as *assigned* and removed from its PWQ.
   c) The worker passes the subquery to its database server instance to execute. The results of this

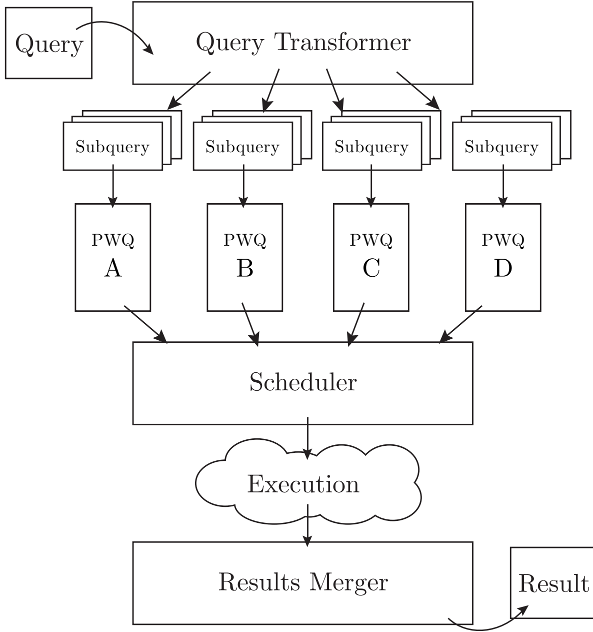

Fig. 5. Upon a query's arrival, Osprey's query transformer divides it up into its respective subqueries, which are then added to the appropriate PWQs. The scheduler assigns outstanding subqueries to available workers (not shown). After the workers execute the queries, results are collected, merged, and finally returned to the user.

query are asynchronously passed back to the query manager. The subquery is marked as *complete*.

3) Once all subqueries have been completed, the results merger combines the subquery results and returns the result to the user.

F. Failure Modes

Failure of the coordinator is a possibility that Osprey does not currently deal with. A checkpointing scheme, where the coordinator persists the subquery results as they come in, has been designed, but we did not describe it because it has not yet been implemented as of the time of the paper. Note that checkpointing at level of worker nodes is *not* necessary because each worker is essentially stateless: a worker can forget about subqueries it has already finished, and any failure during the execution of a subquery will be recovered by the coordinator simply reassigning the task. Note that MapReduce also doesn't support recovery from Coordinator failures.

Because all queries pass through the coordinator, there is a possibility of the coordinator being a bottleneck on query performance. This can occur in two ways: (1) in coordination of the execution of subqueries on the cluster (the workers finish executing tasks faster than the coordinator can assign them or faster than the coordinator can receive them), and (2) the coordinator could be overwhelmed in the task of merging results. We doubt that (1) is a real risk because we expect the assignment of subqueries to be much, much faster than their execution. (2) is potentially a risk, though we assume that subquery responses return few rows (i.e. analytic type queries where pre-aggregation can be pushed down to the workers as we do). If this assumption holds (and our benchmark workloads suggest this is true), this possibility is limited.

We extensively test Osprey and show the results in Section IV; first we discuss how to extend the above execution model to support more general join queries.

G. General Join Queries

Our current implementation of joins in Osprey is limited in that we require that dimension tables be replicated at all nodes. To support queries between dimension tables that do not fit on a single node, or more complex non-star schema joins, some additional mechanism is necessary. We sketch two possible approaches here, similar to those taken in early distributed databases like Gamma [12] and R* [17]. We include these schemes for completeness, but do not evaluate or describe all of the details of these approaches due to space constraints, and because we feel that basic star-schema joins with relatively small dimension tables are the common cases in most data warehouse settings.

To allow dimension tables to be partitioned rather than replicated, several strategies are possible. If the both tables are partitioned on the join attribute, then the strategy described above will simply work, since each node will have the dimension partition that contains exactly those tuples that join with its fact partition. When the fact table is not properly partitioned, one possible strategy is to use *IN-list rewriting*, which is similar to the semi-join technique discussed in the R* paper [17]. The idea is as follows: for each chunk of the fact table stored at a node, run a subquery against the dimension table partitions at each of the remote nodes to return the attribute values from the dimension table needed to compute the join. This subquery includes a clause of the form `WHERE dimension-join-attr IN {value-list}`, where `value-list` is the list of dimension values that are in the chunk of the fact table needed to answer the query. For example, if a node N is running the query `SELECT d1.attr FROM fact,d1 WHERE fact.fk1 = d1.pk AND p` against its local `fact` partition, it would first build `value-list` by running the query `SELECT fk1 FROM fact WHERE p`. Then it would run the query `SELECT attr FROM d1 WHERE pk IN {value-list}` on each of the remote nodes and store these results in local temporary tables. Finally, N would compute the final answer to the query by running a local join between its fact partition and each of these temporary tables. In this way, each of the fact table chunks on a node can be processed independently, just as in the preceding sections.

An alternative to *IN-list rewriting* which may be useful for some general join queries between two large tables is *dynamic rechunking*, similar to dynamic hash repartitioning in Gamma [12]. In this strategy, a new set of chunks is computed on one of the two tables being joined (most likely, this should be the smaller table). Rather creating these new chunks using round-robin partitioning of the tuples on that node, these chunks are computed by hashing over the join

var	purpose
n	total number of workers
m	number of chunks per partition
k	number of backups per partition
l	number of workers artificially stressed
s	stress factor (on scale 0-3)

TABLE I

OSPREY SYSTEM PARAMETERS.

attribute. Then, these new chunks can copied one-at-a-time to the other node, which can compute the join as in *IN-list rewriting*. This strategy has the disadvantage that creating the new chunk tables may be slow, but is superior to *IN-list rewriting* because each tuple from the re-chunked table is sent over the network only once. These new chunks can be used in place of the previous improperly partitioned chunks as long as they are not too highly skewed in size.

Of course, as in traditional distributed databases, when this kind of repartitioning is necessary, it is higher overhead than when joins are properly pre-partitioned. Note, however, that some degree of parallelism is still achieved, as each node can process these subqueries in parallel. Executing such repartitioning joins through a middleware layer is one case where Osprey will likely perform worse than a non-middleware implementation of a distributed database, which can pipeline the flow of tuples from remote sites into local join operators without materializing intermediate results into temporary tables. An implementation of Osprey-like techniques inside of a database (rather than in middleware) would be able to take advantage of this kind of pipelining.

IV. PERFORMANCE EVALUATION

Osprey was tested using a data warehouse running on the SSB benchmark [3]. Testing for Osprey breaks down into two suites: system-level tests, to demonstrate the scalability of the system, and load balancing tests, to test our fault tolerance claims.

We first explain our experimental setup and the test data and workload. We then present scalability results, followed by load balancing results.

A. Experiment Setup

We use several parameters (shown in Table I) to explore the performance of Osprey. The stress factor l and s are discussed further in Section IV-A.5.

In addition to system properties like those listed above, we chose to vary the set of queries run. In most cases, we run the full set of 13 SSB queries; in some others, however, shorter representative sets of queries were run instead.

1) Platform: We ran Osprey across a network of 9 commodity workstations – 1 coordinator machine and 8 workers. Each contains two Intel Pentium 4 3.06 GHz CPU, 2 GB of main memory, and Debian 4.0, Linux kernel 2.6.27-9-server. Each machine ran a remotely-accessible Postgres 8.3 server, with a limited buffer pool (512 MB) to help limit the effect of caching. All machines were connected via a gigabit-Ethernet switch.

2) SSB Test Data: Our experiments used the *Star Schema Benchmark*, Scale Factor 10; SSB is a derivative of TPC-H[2] [3]. The fact table in SSB is `lineorders` (a merging of TPC-H's `lineitem` and `orders` tables). Osprey horizontally partitions this fact table and distributes it to workers. SSB also includes 4 dimension tables. SSB specifies no indexes or integrity constraints on the database besides identifying primary keys on the dimension tables. We discuss our indexes used in the test cases more completely in Section IV-A.4.

SSB (and, in turn, TPC-H) is not the most general schema or query workload, but is designed to represent typical data warehouse demands – TPC-H is the most widely used data warehousing benchmark. Our assumptions about the selectivity of filters of the queries (see Section III-F) and our fact-table based approach were sufficient for SSB, which is suggestive that Osprey's query and schema restrictions are not too limiting to the applications for which it was designed.

We used SSB to generate 5.5 GB of data: a fact table of 5.4 GB (60,000,000 tuples), and dimension tables totaling 105 MB.

3) Query Workload: The full set of SSB queries consists of four "query flights," of three to four queries each, for a total of 13 queries. Below are two sample queries of varying complexity from the SSB query set.

```
Query 1.1:
SELECT SUM(lo_extendedprice*lo_discount)
    AS revenue
  FROM lineorder, date
  WHERE lo_orderdate = d_datekey
    AND d_year = 1993
    AND lo_discount BETWEEN 1 AND 3
    AND lo_quantity < 25;
```

Query 1.1 is the first query in the set and returns a single row – it performs an aggregate over the entire table, filtering with selectivity 0.019, producing 1.14 million tuples at Scale Factor 10. Query 4.3, on the other hand, is the last query in the set, returns the fewest rows of all test queries. Its filters have a selectivity of 0.000091.

```
Query 4.3:
SELECT d_year, s_city, p_brand1,
    sum(lo_revenue - lo_supplycost)
    AS profit
  FROM date, customer, supplier,
      part, lineorder
  WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_partkey = p_partkey
    AND lo_orderdate = d_datekey
    AND s_nation = 'UNITED STATES'
    AND (d_year = 1997 OR d_year = 1998)
    AND p_category = 'MFGR#14'
```

[2] The TPC-H schema is not a clear star schema because the largest table, `lineitem`, is only four times larger than the next largest table, `orders`, which is why we chose not to experiment with it.

s	# CPU threads	# VM threads	# HDD threads
0	0	0	0
1	5	5	5
2	6	6	10
3	∞	∞	∞

TABLE II

MAPPING OF STRESS SCALE s TO ACTUAL STRESS UTILITY PARAMETERS. $s = 0$ CORRESPONDS TO NO LOAD; $s =$ CORRESPONDS TO "INFINITE LOAD" (WORKER DEATH). COLUMNS CORRESPOND TO THE NUMBER OF THREADS SPAWNED TO CONSUME SYSTEM RESOURCES. CPU THREADS SPIN ON A FLOATING-POINT COMPUTATION, VM THREADS ALLOCATE AND HOLD MEMORY, AND HDD THREADS CONTINUOUSLY WRITE AND DELETE DATA FROM THE HARD DRIVE.

```
GROUP BY d_year, s_city, p_brand1;
ORDER BY d_year, s_city, p_brand1;
```

SSB queries were designed to span the range of typical data warehousing queries in the commercial space. Queries' selectivity and functions are designed to be as varied as possible, attempting to provide TPC-H-like coverage of the data set. Unless otherwise noted, our tests involve the serial execution of the entire SSB query set, with time reflecting the total runtime of all queries.

4) Indexes and Optimization: One of the main advantages of Osprey over computation frameworks like MapReduce is the ability to take advantage of features inherent in a relational database – for example, indexes and clustering. Because we expect Osprey to be used with a largely read-only load, indexes were the clear choice to optimize performance. We created indexes on all commonly filtered fields for both the fact and dimension tables on all machines (each machine indexed its local copy of its data).

The fact table – the tables on each worker representing fractions of the fact table – was then clustered around the field `lo_orderdate`, as almost all queries in the SSB query set were restricted by some date property. Finally, `ANALYZE` was run on all individual tables to provide accurate histogram data for Postgres' query planner.

5) Stress: Since Osprey is designed for load balancing, we need a way to artificially load a given machine, in a controlled and repeatable way. We adapted a system stressing utility, CPU Burn-In [18], which spawns a number of threads to consume system resources, to suit our purposes. The utility spawns the following types of threads:

1) a CPU-hogging thread, which executes `sqrt(rand())` in an infinite loop
2) a virtual-memory-hogging thread, which allocates a certain amount of memory and sleeps - by default, 256 MB
3) a hard drive-hogging thread, which continuously writes data (in our case, 256 MB) out to disk

We scaled the amount of stress by setting the number of these threads to spawn, in addition to other parameters of the threads' behavior. We feel this artificial load is a reasonable approximation of real stress in a system.

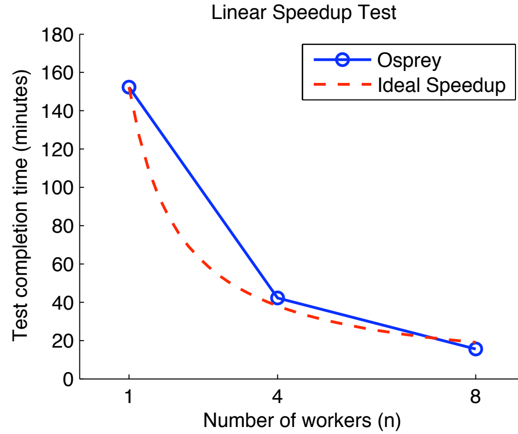

Fig. 6. Linear speedup test of Osprey. (Control parameters: $k = 1$, $m = 10$.) Osprey demonstrates linear speedup. The dashed line shows the ideal linear speedup curve. Control parameters: $k = 1$, $m = 10$, no stress.

For convenience, we define a four-point stress scale s, where $s = 0$ represents no artificial stress and $s = 3$ represents worker being "infinitely loaded." (In our experiments, workers stressed to $s = 3$ were simply disconnected from the Osprey network, rather than actually overloading them.) Table II maps the other s values to actual parameters of the stressing utility.

Table values were determined empirically, by observing the effect a set of parameters had on a controlled execution, and retaining the set that provided our desired consequences under the controlled environment.

B. Results

In this section, we discuss our scalability, overhead, and fault-tolerance results.

1) Scalability: Figure 6 summarizes the results of our speedup test; our goal here is to demonstrate *linear speedup*, where n machines complete a query n times faster than a single machine. For a fixed 6 GB data set, we increase the number of workers n in the cluster and measure completion time for the full test query set. The backup factor k is set to 1 and $m = 10$ chunks per partition. The dashed line shows the ideal linear speedup curve. The results clearly show linear speedup.

Our system is inspired by some MapReduce techniques, but does not necessarily imply that it is MapReduce scale; our tests used (only) up to 8 nodes but we felt this was reasonable, as database clusters of 8–12 machines are common. Commercial deployments of Oracle RAC typically consist of 2–4 machines, and other commercial vendors report customers using clusters of 8–16 machines.

We also note that, with a replication factor of $k = 1$, the system is handling almost 12 GB of data. With 8 worker machines in the cluster, each node is handling 1.5 GB – which exceeds the buffer pool size allotted to each Postgres instance by a factor of 3. This should mitigate any in-memory caching effects, suggesting the system will scale for other, larger workloads.

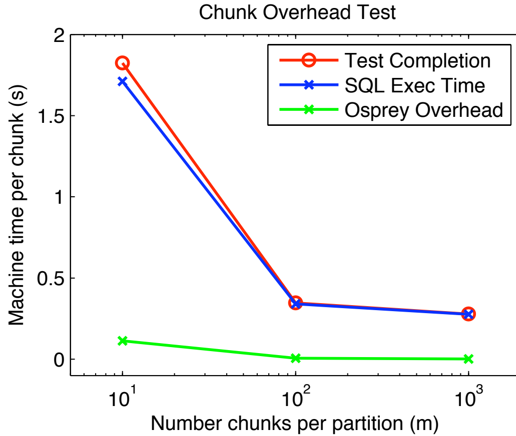

Fig. 7. Overhead test of Osprey. Here we vary the number of chunks per partition, to test the overhead introduced by increasing the number of chunks. This figure compares the total Osprey query execution time ("Test Completion"), the total time the average worker spent just in the Postgres instance ("SQL Exec Time"), and the resulting difference in overhead ("Osprey Overhead"), averaged over the number of chunks. Control parameters: $n = 4$, $k = 1$.

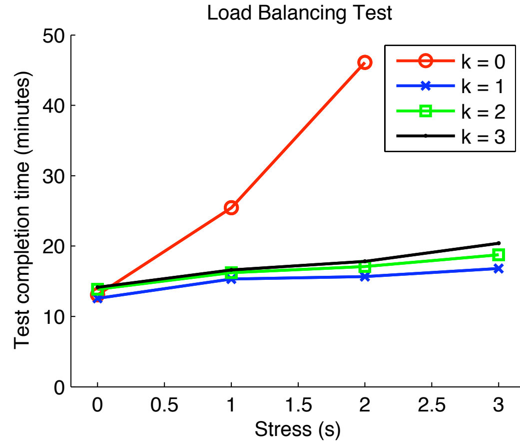

Fig. 8. Load balancing test of Osprey, in which one worker is artificially stressed. Osprey achieves almost linear load balancing, even when one machine is unavailable ($s = 3$). Note that for $k = 0$, $s = 3$ has no corresponding time – with 1 worker down and no backups, no query could complete. Control parameters: $n = 4, m = 1000, l = 1$.

2) Overhead: One of the challenges in a distributed system is to minimize the overhead of scheduling computation. Middleware approaches such as Osprey are particularly problematic in terms of overhead because low-level optimizations (inside the DBMS) are not available.

There are 3 primary sources of overhead in Osprey:

1) Subquery scheduling,
2) Delivering subquery requests to worker database servers and returning the results, and
3) Startup and tear-down costs of the actual execution of the subquery on each worker database server.

Figure 7 summarizes the results of our test. We vary the number of chunks per partition m and measure the time spent computing on each chunk (averaged over all chunks). The "Test Completion" curve shows the total Osprey query execution time, from when the query was started to when the merged result was returned to the user, averaged by the number of chunks. The "SQL Exec Time" curve shows the time spent processing each chunk in just in each worker's Postgres instance (averaged over the chunks). We see that the average time Postgres takes per chunk drops significantly between $m = 10$ and $m = 100$, as the size of each chunk has shrunk by a factor of 10. But we further see that the time per chunk time does *not* drop between $m = 100$ and $m = 1000$, showing that the chunks are so small that fixed startup costs of executing a query in Postgres dominate.

Finally the "Osprey Overhead" is the difference between the two other curves. This shows the per-chunk overhead in Osprey due to scheduling and dispatching subqueries (and receiving the results from the workers). The overhead at $m = 10$ is larger than at $m = 100, 1000$, which reflects the fact that Osprey has fixed startup costs independent of the number of chunks. Because there are fewer chunks to amortize this cost over, the overhead per chunk at $m = 10$ is larger. The amortized overhead becomes very small as m increases.

We see clearly that we cannot make m too large – as chunks become smaller and smaller, the fixed cost for executing a subquery in Postgres dominates, which gives bad overall query performance. We also cannot make m too small either, because the number of chunks per partition controls how granular our load balancing can be. There is a tradeoff in Osprey between query performance (small m) and load balancing (large m); it appears that m values between 10 and 100 are a good choice for our configuration.

3) Load Balancing: Osprey's use of chained declustering and dynamic work scheduling was intended to provide resilience to worker failure or uneven load across worker machines. As discussed in Section III-D.1, chained declustering provides a way for excess work to cascade to its neighbors.

We ran two experiments using a reduced query set that repeats Query 1.1 five times:

1) Tests of the load balancing as we vary the amount of stress on one machine, and
2) Testing the ability of the system to load balance as we stressed more and more machines.

Figure 8 shows our results as we vary the amount of stress placed on $l = 1$ machine, for varying backup factors k. With $k = 0$, no backups are used, so no load balancing can occur.

For any other value of k, the expected load balancing curve is linear. If the time when no node is stressed (s=0) is t_0, then with $s = 3$, the expected completion time is $t_0 \cdot n/(n - l)$, where $n/(n - 1)$ represents the overall increased amount of work per worker due to the single worker's failure. We see that Osprey performs as expected, with runtime climbing from 11 seconds at $s = 0$ to about 14 seconds at $s = 3$, which is close to the expected slowdown of 4/3. This shows that Osprey is able to quickly accept load from a stressed or failing node.

Next, Figure 9 shows our results as we vary the number of machines stressed l, for a fixed stress level $s = 1$ over various settings of k. Once again, with $k = 0$, no load balancing can

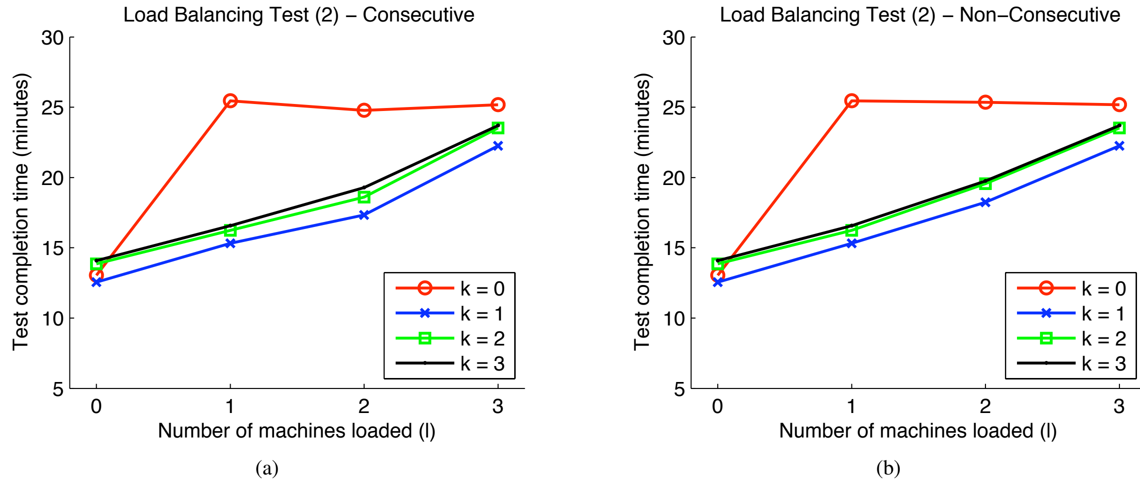

Fig. 9. Load balancing test of Osprey. The number of machines experiencing stress is varied against the number of backups. (a) shows when $l = 2$ adjacent machines (that share a partition) are stressed. (b) represents the non-consecutive $l = 2$ case. Osprey shows good load balancing results in both cases. Control parameters: $n = 4$, $m = 1000$, $s = 1$.

occur, so the total completion time is the maximum completion time of each of the workers. So for $l = 1 \ldots 3$, the completion time is constant. As in the first test, the ideal load balancing curve is linear with respect to the number of machines loaded. We see that Osprey performs as expected.

It is important to note that for $l = 2$ in a cluster of $n = 4$ machines, we have a choice of loading consecutive or non-consecutive machines. This distinction is necessary because in chained declustering, if $k + 1$ consecutive machines fail, then data is unavailable because a partition is stored on $k + 1$ consecutive machines. But for any load short of actual failure, a good dynamic scheduler should attempt to keep all workers equally busy, whether or not stressed machines are adjacent.

Figures 9(a) and 9(b) show Osprey's performance when the $l = 2$ machines are consecutive and non-consecutive, respectively. (Results for $l = 0$, 1, and 3 are the same for both graphs.) We see that Osprey's load balancer performs the same for both choices, with (as expected) performance weakening as more machines are loaded.

Finally, Figure 10 shows the load balance for a cluster of $n = 3$ machines over time. The y-axis is the fraction of total computation each worker contributes – if the system were perfectly load balanced, all workers should contribute equally. We see that as the system starts, the three workers hover around 0.33, meaning they are balanced. At $t = 190$, we stop Worker 1 – the remaining workers quickly compensate for Worker 1's loss, settling into a new equilibrium where each worker contributes 0.5 of the total work.

We note that in these experiments (a) we started with an empty buffer pool, (b) we attempted to avoid in-memory effects by limiting the size of the buffer pool to 512 MB and, (c) stressing ($s > 0$) also consumed memory, disrupting in-memory/caching benefits. Furthermore, with replication factor $k > 0$, the actual data size is much bigger than 5.5 GB. For these load balancing tests, we ran a cluster with 4 workers. With a replication factor of 3, each worker is responsible for

Fig. 10. Load balancing test of Osprey, shown over time. The y-axis is the relative fraction of total computation time each worker contributes. At $t = 190$, we cause Worker 1 to fail. Both before and after the event, each worker is contributing its fair share to the total computation time.

4.2 GB of data, which is larger than each machine's 2 GB of physical memory.

4) Scheduling Algorithms: Osprey implements three schedulers, *random*, *LQF*, and *majorization*. Figure 11 shows Osprey's results for each scheduling algorithm, in which $l = 2$ non-adjacent workers were stressed with varying settings of s. The query set was Queries 1.2, 2.3, 3.4, and 4.2, selected because they accessed a wide range of rows in the fact table and were a representative sample of the full SSB test suite.

With no stress, all the schedulers should do roughly the same. The random scheduler predictably does the worst, running about 10% slower than LQF once stressed. Majorization actually runs slower than LQF – it is a much more complicated scheduler, having to look at the state of all PWQs before making an assignment; this scheduling overhead hurts majorization's performance relative to LQF.

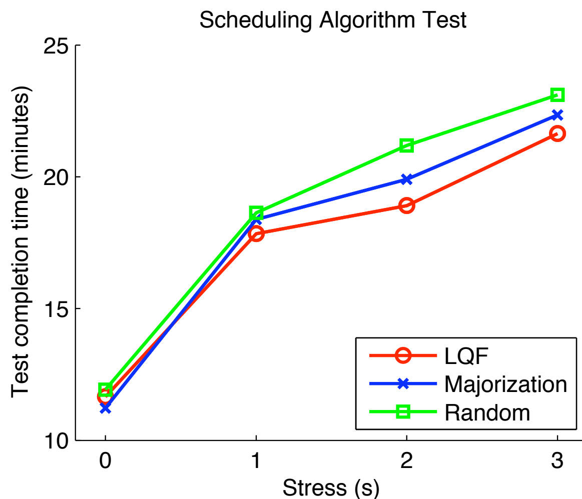

Fig. 11. Comparison of different scheduling algorithms. *Longest Queue First* chooses a subquery from the longest PWQ. *Majorization* minimizes the differences in PWQ lengths. *Random* chooses a subquery at random from available PWQs. Control parameters: $n = 4$, $k = 2$, $l = 2$.

V. Conclusions and Future Work

We presented Osprey, a middleware implementation of MapReduce-style fault tolerance for a SQL database. We have demonstrated linear speedup for Osprey and that the overhead for running a query is acceptable. We have further shown that Osprey provides good load balancing and fault tolerance properties, based on several key ideas:

1) Splitting up queries into subqueries that can be executed in parallel,
2) Dynamic scheduling of those subqueries' execution on the workers, and
3) Re-execution of straggler subqueries.

These strategies are adapted from similar strategies employed in MapReduce, but Osprey differs in that it uses SQL-based database systems, obtaining the benefits of indexing, constraints, query optimization, and recovery. It is also employs shared-nothing design using chained declustering, rather than a distributed file system (GFS) available to all machines.

We implemented Osprey as a middleware layer so that we could have a usable distributed database system without rewriting the query optimizer, executor, parser, etc. We believe that existing distributed database vendors could take advantage of the ideas in this paper if they were to internally partition tables into chunks to facilitate mid-query restart. Though adapting an existing system would be a substantial amount of work, it may be necessary as database vendors have increasing pressure to scale their systems to every larger datasets and clusters. Exploring the impact of our chunking approaches on the performance of distributed queries (versus existing distributed databases that do not perform such chunking) is an interesting area for future work.

There are several other future areas of work we would like to explore: (1) *Implement updates, inserts, and deletes* – as proposed in Section III-C.2. (2) *More advanced query schedulers* – caching of chunks in the worker's memory is ignored during our scheduling, but assigning a chunk to a worker that already has it in memory could improve the overall query performance. (3) *More accurate measure of chunk work.* Osprey's scheduling algorithms assign work based on the *number* of subqueries remaining in each PWQ. Ideally, the length of each PWQ is the computation time remaining for that partition, for which the number of subqueries remaining is a proxy; more robust load balancing results should be possible with an accurate remaining-time estimate, and (4) *evaluation of general join strategies* as described in Section III-G.

VI. Acknowledgments

This work was supported by NSF Grant CLuE-0844013.

References

[1] H.-I. Hsiao and D. DeWitt, "Chained declustering: a new availability strategy for multiprocessor database machines," in *Data Engineering, 1990. Proceedings. Sixth International Conference on*, Feb 1990, pp. 456–465.
[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004, pp. 137–150.
[3] P. E. O'Neil, E. J. O'Neil, and X. Chen, "The star schema benchmark (ssb)," *http://www.cs.umb.edu/p̃oneil/StarSchemaB.PDF*, January 2007.
[4] "Greenplum," *http://www.greenplum.com/*.
[5] "Aster," *http://www.asterdata.com/*.
[6] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," in *Proceedings of VLDB*, 2009.
[7] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears, "Boom: Data-centric programming in the datacenter," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-113, Aug 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-113.html
[8] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, 2005.
[9] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of SIGMOD*, 2005.
[10] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, Dec. 2007, pp. 290–297.
[11] F. Pedone and S. Frølund, "Pronto: High availability for standard off-the-shelf databases," *J. Parallel Distrib. Comput.*, vol. 68, no. 2, pp. 150–164, 2008.
[12] D. J. Dewitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. i Hsiao, and R. Rasmussen, "The gamma database machine project," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 44–62, 1990.
[13] L. Golubchik, J. Lui, and R. Muntz, "Chained declustering: load balancing and robustness to skew and failures," in *Research Issues on Data Engineering, 1992: Transaction and Query Processing, Second International Workshop on*, Feb 1992, pp. 88–95.
[14] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[15] P.-Å. Larson, "Data reduction by partial preaggregation," in *ICDE*, 2002.
[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *J. Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997. [Online]. Available: citeseer.ist.psu.edu/gray97data.html
[17] L. Mackert and G. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *Proceedings of VLDB*, 1986.
[18] M. Mienik, "CPU burn-in," *http://users.bigpond.net.au/CPUburn/*.