Fault tolerance approach for OLAP

Dalhousie University
Department of Computer Science
Halifax, NS

Hatem Nassrat
nassrat@cs.dal.ca
B00393388
December 2006
Parallel Computing

Contents

Fault tolerance approach for OLAP

Abstract

OLAP is a widely used approach that allows quick answers to analytical queries that may lie in a different dimension than the traditional data view. Since OLAP systems deal with large sets of data, many of them are set up either in parallel or in a distributed manner (such as a grid of computers). Parallel and distributed machines tend to use standard disks to store their data and some may often fail. This paper will aim to address data fault tolerance in such systems, with a model integration in one of the discussed fault tolerance schemes.

Hatem Nassrat

Dalhousie University - Dec 2006

Parallel Computing - A. Rau-Chaplin

Introduction

OLAP today is being used by many business enterprises, in order for them to be able to make decisions based on the data analysis. Data in OLAP contains many dimensions as each field in a relation represents such a dimension. These dimensions from large data structures that are used by the OLAP system in their analytical queries, and today are being computed, partitioned and stored on parallel machines. Many parallel machines, such as shared-nothing clusters, tend to have personal computing architectures with personal computing hardware integrated in their units. The use of such machines allows, with a larger probability, for failures. Therefore focus in the field of fault tolerance should be taken when designing such systems.

There has been many discussions in this area, with many different designs and techniques on making parallel systems fault tolerant [1]. These methods can be directly applied in a parallel OLAP model, to gain the desired fault tolerance. This paper aims to discuss fault tolerance issues of the data stored by OLAP systems, in a quest to develop a parallel model to efficiently survive multiple failures in various parallel machines.

Notations

The notations used here are similar to the ones found in [2]. In our further discussion we will be analyzing a *p+1-node* (where the extra node, is a management node (described in the design bellow)) shared nothing cluster, that will allow for a maximum of k node failures (being k-fault tolerant). When looking at a failed system, $p' < p$ will be used to refer to the remaining nodes and f will be used to refer to the failed nodes.

Failures in a Parallel System

There are different models associated with certain failures of parallel systems, and help determine the behavior of the system after the fault. Fault tolerance techniques would assume a certain model in order to explain the types of faults it can handle [1]. The general faults are to be described in this section with an emphasis on fail-stop faults, the type to be tolerated by our proposed system.

Byzantine fault model represents the most complicated type of fault to handle. In this model failed nodes are still alive in the system, interacting with the rest of it as if not faulty. These nodes tend to give incorrect output (thus the fault) and requires complicated techniques in order to determine which nodes have actually failed. This model can represent random system failures and more importantly malicious hacker attacks and random memory corruption [1]. It is known that no guarantees can be made when m or more failures are experienced in a system with 3m+1 nodes [3].

The main model that we will aim to tolerate using our approach would be **Fail-stop** faults. As the name implies, this model states that when a failure is experienced by a node, it ceases to produce output and stops interacting with the rest of the system [1]. Furthermore, the other nodes should be able to identify the failed node(s). This model mainly represents failures such as a node crash.

A third model can also be used in classifying faults, and is known as **Fail-stutter** faults. This model is not as general as the Byzantine model; However, is an extension to the Fail-stop model. It includes all the provisions of the fail-stop model, but also allows for weak performance faults. Such a fault is one where a component unexpectedly provides low performance within the parallel system, however continues to function properly with correct output. This model may be important in parallel systems especially since the speed of the system is related to the speed of the slowest component. This model allows for faults such as poor latency performance of a network switch, or high traffic loads in a sub section of the network. This model is still not commonly used in the community despite its advantages [1].

In general, using any fault model, there are certain requirements that need to be fulfilled in order to fulfill fault tolerance. They are that failures can be detected, information needed to continue the computation is available and the computation can be restarted [4].

Fault tolerance

Many parallel computing machines depend on centralized components to function properly [1]. Management nodes tend to handle job scheduling and node monitoring, storage nodes provide access to disk storage and compute nodes tend to perform the computations. Clusters also have head nodes which allow users to interact with the parallel machine without slowing down the other nodes. In a parallel machine designed for OLAP many of these components are not essential. The diagram in Figure 1 shows a design for a parallel shared nothing cluster that may be used to implement OLAP. The management node is responsible for receiving the queries that are to be processed. This node then sends the query to all query nodes. The query nodes answer the query based on the data stored at that node. The results are then sent to the management node to return to the sender of the query. This design needs to be complemented with a strategy of dividing the data amongst the nodes. The data should be partitioned such that queries may be answered as fast as possible on the parallel machine, meaning that each node should have the same response time on average for a given query. A method of doing so is partitioning the data the records in the different OLAP views using Hilbert curves. This method is proven to perform well in certain implementations. For our discussion of fault tolerance, it is sufficient to show a fault tolerant strategy for an arbitrary view of an OLAP table, with its records partitioned across the parallel machine. Also the design shown in the figure above would be used as a basis of the fault tolerant strategies to be discussed.

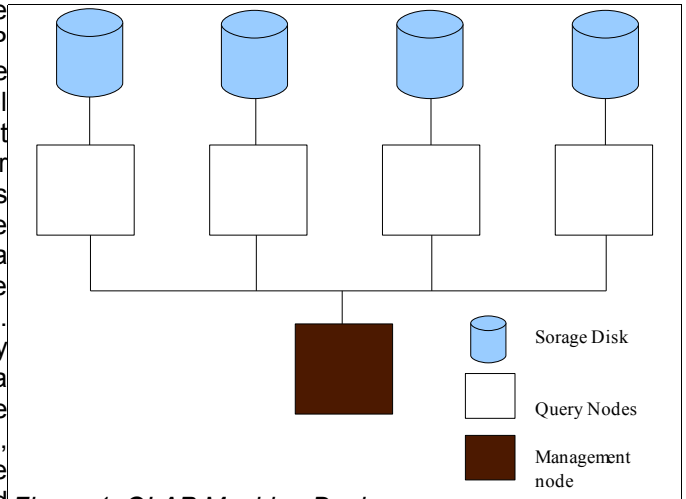

Figure 1: OLAP Machine Design

Replication

Fault tolerance in a parallel machine may be done by means of replicating fault prone components. In the field of fault tolerance such replication can take one of two forms [1].

The first being active replication [1], where nodes are cloned to form backup nodes. The backup nodes are running along side the standard nodes, receiving all input sent to them, thus always being up to date. When an unexpected behavior by the standard nodes is observed, the backup nodes promote themselves to becoming the standard nodes and take over the responsibilities. This change would require negligible time since the nodes are in an identical state. An extension to this approach that is able to handle byzantine faults, requires multiple backup nodes, running in parallel, with a consensus approach. All nodes would compute the output for a given input and a byzantine algorithm would be used to vote on which is the correct output. Nodes failing multiple times would be deemed faulty. The standard approach does not only handle a single fault in each of the nodes, it also would be quite expensive for many systems. The systems would have to have twice as many nodes, as they would have originally and not be able to use these machines independently. The extended approach leads to even more redundant machines. In certain environments where errors cannot be tolerated the extended approach aims to solve the byzantine generals problem.

Along side active replication, is another approach known as passive replication [1]. In such an approach “cold spares” are used to employ all the software installed on the primary nodes. The “cold spares” should be kept up to date via checkpoints in the system in order to not incur any interruption of service. Using a monitoring approach, a faulty node can be detected and replaced with its cold spare. In a finer approach fault prone node components can be replicated within a node, and switched to when that part is considered faulty. In both cases, the systems administrator would be notified with the fault to debug the original system, or replace it. Both these replication approaches would not be adequate for OLAP, since they do not tolerate many faults (especially subsequent ones) and may result in loss of data with an unknown probability.

Redundancy & Erasure Codes

Coded redundancy can be used to design a more OLAP oriented fault tolerant scheme. In coded redundancy the data is encoded to form the backups. To be able to recover data after k out of p nodes have failed, then the amount of information available in the remaining k-p nodes must be equal to the original data [2]. One can see that this requires storing multiple (redundant) copies of the data, this increases the amount of data needed to be stored on each node. In general, using coded redundancy a moderate net storage cost of 2.5 times will allow for a failure rate of 33% [2].

Using erasure codes, an optimal solution appears for our OLAP problem. Reed-Solomon codes are an example of erasure codes that allow for a data fault tolerant machine. Many applications such as compact disc players use such codes to be able to correct data that has been disturbed due to noise in the channel. This coding scheme, published in 1960 [5], uses finite field geometry to code and decode data. Using such codes, data can be divided into an arbitrary number of pieces m, and can be set to withstand k faults. Meaning that only m-k units of the backups are needed in order to store the data. To apply this to our OLAP design, each query node would use RS codes, to backup its partition of the OLAP records. The backups would consist of p-1 units, to be distributed across the p processor machine, and would be computed to tolerate k faults. In such a scheme, when any k nodes fail, the data can be decoded from the remaining p' (p' = p-k) nodes, since the computed backups would allow such a recovery.

The way these codes work, is that if we require p-1 chunks of data to tolerate k faults, then the original data would be divided into (p-1) – k units (referred to as n in [6]), and k checksums (referred to as m in [6]). Thus having n+m chunks in total (equal to p-1) to be distributed for backup purposes. These checksums allow for the fault tolerant properties of these codes, as they can be used to recompute the original data, when enough surrounding data is present. This condition is satisfied when and n or more chunks (≥p-1-k) are available.

Computation of RS codes requires the implementation of certain mathematical functions. Many of the functions used in these computations are arithmetic functions in the Galois Field. A mathematically optimized implementation of Galois Field arithmetic (with RS examples), is available under the GNU Lesser General Public License on the web [6]. Along with an excellent tutorial on the coding theory needed to compute Reed Solomon codes [7]. The GF library, found in [6] allows for GF arithmetic to the fields GF(2^8) and GF(2^{16}). Where 8 and 16 represent the length of the blocks (words) to be coded. This library computes tables (described bellow) to aid in optimizing the coding. The data structures created in the GF(2^8) field, are smaller than the ones formed for the larger field. However, in the smaller field only 255 chunks are allowed including the checksum chunks (i.e. The number of partitions x + the number of checksums y = p-1 in our OLAP system ≤ 255). The GF(2^{16}) field allows up to 65536 chunks. In a large scale distributed OLAP environment the need for the larger field may be explainable; However, in most scenarios where less than 255 copies are needed, i.e. a machine with at most 256 query nodes (in the design proposed above) would fit in the smaller, more optimized field. This parameter is passed to ***gflib*** [6] as compiler directives.

1. Choose a value of w such that $2^w > n + m$. It is easiest to choose $w = 8$ or $w = 16$, as words then fall directly on byte boundaries. Note that with $w = 16$, $n + m$ can be as large as $65,535$.
2. Set up the tables `gflog` and `gfilog` as described in Appendix A and implemented in Figure 4.
3. Set up the matrix F to be the $m \times n$ Vandermonde matrix: $f_{i,j} = j^{i-1}$ (for $1 \leq i \leq m, 1 \leq j \leq n$) where multiplication is performed over $GF(2^w)$.
4. Use the matrix F to calculate and maintain each word of the checksum devices from the words of the data devices. Again, all addition and multiplication is performed over $GF(2^w)$.
5. If any number of devices up to m fail, then they can be restored in the following manner. Choose any n of the remaining devices, and construct the matrix A' and vector E' as defined previously. Then solve for D in $A'D = E'$. This enables the data devices to be restored. Once the data devices are restored, the failed checksum devices may be recalculated using the matrix F.

Figure 2: Algorithm to create n+m chunks from the data, that can be recovered after m chunks being lost

The algorithm seen in Figure 2, is the general algorithm for computing the coding of the data into chunks

of n+m = p-1. The *gflog* and *gfilog* (described in step two (Fig 2)), are lookup tables, the first being the logarithms of the table indicies in the respective Galois Field, while the second is the inverse logarithm in the field. Thus maintaining the following relation: "`gflog[gfilog[i]]=i` , and `int gfilog[gflog[i]] = i`" [7].

A design similar to the one discussed above, is analyzed in [7] as a raid like configuration. They have analyzed the time needed to perform the encoding of n+m blocks of size S_{block} (Fig 4) in such a system. The formula for the time complexity can be seen in Figure 4. In the Figure R_{XOR} represents the rate of an XOR operation and R_{GFmult} is the rate of performing a multiplication in GF. The reason for having one less GF multiply operation, is due to the fact that the first checksum is computed without multiplication, since the first row of the matrix used for the multiplication is all ones. The formula shows that for each word around m XOR and GF multiplication operations occur to calculate its m checksums.

Another operation that is useful is overwriting words in a file. This can be done with a certain degree of efficiency with RS codes. The cost of overwriting a word in one of the files is captured in the formula presented in Figure 5. To overwrite a word, the word has to be written to the backup containing the original word + the corresponding word in each of the checksums has to be recomputed (hence the number m+1 in the formula). The variable j represents the partition in which the original data is located. If the data is located in the first partition, the computation would be faster than if the word to be changed was in any of the other segments.

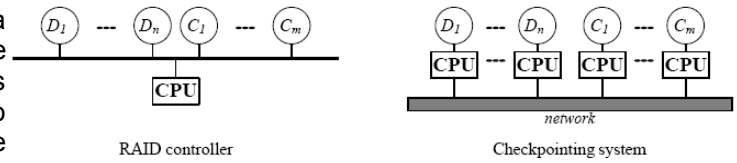

Figure 3: Raid like configurations [7]

$$S_{Block}(n-1)\left(\frac{m}{R_{\text{XOR}}}+\frac{(m-1)}{R_{GFmult}}\right)$$

Figure 5: Time to compute n+m blocks [7]

$$\text{The cost of writing one word to } m+1 \text{ disks} + \begin{cases} \left(\frac{m+1}{R_{\text{XOR}}}\right) & \text{if } j=1 \\ \left(\frac{m+1}{R_{\text{XOR}}}\right)+\left(\frac{m-1}{R_{GFmult}}\right) & \text{otherwise.} \end{cases}$$

Figure 4: Time to overwrite one word in segment j [7]

The calculation required for recovery has two parts, a Gaussian Elimination (taking O(size(f)^{2}n)) and recalculation [7]. Since the number of failed nodes is relatively small the first step is compared to linear speed. The time to recover a block of data is upper bounded by the formula seen in figure 6. In Figure 6 the size(f) (number of failed nodes) is referred to as y. According to [7] the cost of both updating and writing is dominated by the time take to write to the disks.

$$\left(\begin{array}{c}\text{The cost of reading one}\\ \text{block from each of } n \text{ disks}\end{array}\right)+\left(\frac{(y)S_{Block}(n-1)}{R_{\text{XOR}}}\right)+\left(\frac{(y)S_{Block}(n)}{R_{GFmult}}\right)+\left(\begin{array}{c}\text{The costs of writing one}\\ \text{block to each of } y \text{ disks}\end{array}\right)$$

Figure 6: Time to recover one block [7]

These are the redundancy costs of a raid like system. Other systems are proposed in [7] and are labeled check pointing systems. Broadcast and Fan-out algorithms are analyzed for such systems and are analyzed for time complexity within [7]. The paper also discusses Hamming codes as an optimal alternative to Reed Solomon Codes when $k \in \{1,2\}$.

MPI Applications

A common misunderstanding amongst MPI programmers is that "MPI is not fault tolerant" [4]. This misconception is derived from another misconception that if any process dies, then the rest of the processes must die. This occurs when the communicator MPI_COMM_WORLD is used since the default error handler on that communicator is MPI_ERRORS_ARE_FATAL. Using this error handler, if a process exits before sending an MPI_Finalize, then the

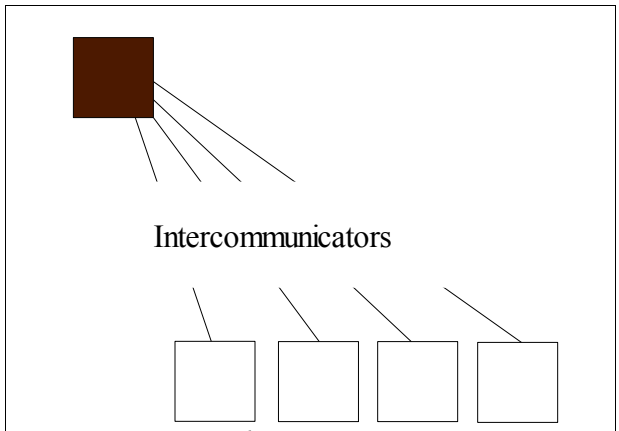

Figure 7: No worker group communication

rest of the processes should detect this condition and exit under MPI specifications [4]. The MPI specification defines this as the default; However, this default behavior can be changed.

The MPI specification mandates that all communicators inherit their error handlers from the parent communicator, and thus any communicator created from MPI_COMM_WORLD would have the ERRORS_ARE_FATAL error handler. To change this one can change the error handler on the communicator to be MPI_ERRORS_RETURN [4], thus allowing MPI programs to return with an error code instead of exiting. Also communicators used to communicate with nodes that have failed should be abandoned. These procedures are explained in detail in [4] with samples from MPI-1 in C and Fortran. In an OLAP setting these characteristics can be appended to the model earlier proposed in this paper. The Management node would handle the communication with each of the Query nodes, and detect a failure at a specific node when their communication fails (i.e. An error code is returned by an MPI function).

Some MPI libraries such as MPI-FT aimed to create a fault tolerant framework for MPI applications. Under such implementations of MPI, checkpoints and message logging are used to restart aborted processes. However, many believe that fault tolerance cannot be a property of an MPI implementation since no specific implementation can ensure that a system is immune from all faults. The claim is that fault tolerance is a property of an MPI program, and should be designed according to the users needs [4]. [4] also has a proposition to encapsulate code that would allow programmers to set up an MPI applications in such a way as the one illustrated in Fig 7. Under the setup discussed above (Figure 7) along with RS coding, certain responsibilities have to be carried out by the management node while others by the query node. In the table bellow are the different responsibilities that can be carried out by the different types of nodes. It must be noted that in such a scheme in MPI the management node cannot tolerate faults, since it is responsible for co-ordinating the Query nodes. The query nodes will be designed to tolerate *k* disk failures.

Management Node's Responsibilities	Query Nodes' Responsibilities
● Receive RS Backups OR Receive Data & Compute Backups ● Distribute backups amongst nodes (p-1 parts to each node) ● Detect dead nodes and abandon communicator ● Recover Dead nodes Data and distribute ● Send Query and Receive results from all nodes	● Send Computed Backups OR Send Data ● Receive (p-1) backup segments ● Receive extra records from management node to be appended to nodes records (for query processing purposes) ● Process Queries from and send results to manager node.

OLAP FT Algorithm

Using the different aspects discussed above, a general view of the procedures in the fault tolerant OLAP design can be generalized into the algorithm seen bellow. Note that when recovering the data from the remaining *p'* nodes, after having already experienced a failure previously, would result in tracking which records were backed up. The algorithm below, first determines that there are enough pieces to recover the data, then deletes all the records received at any recovery stage from live Query nodes. Following that the entire set of records are recomputed from backups and distributed.

General Algorithm

Precondition: Data distributed among the Query Nodes
Initialization:
    For all nodes compute backups and send to manager
    Manger: distribute backups amongst query nodes
Processing:
    Answer queries from management Node
After any Failure:
    if size(*f*) < *k* :
        for each *p'* drop all records not in initial partition

```
    for all failed nodes recover their data from the backups at p'
    # this ensures all records are recovered
    Distribute recovered backups to the p' nodes
else:
    Exit("No more than k failures can be tolerated")
```

The appendix includes more MPI-oriented C style pseudo code representation of the above algorithm, that includes aspects discussed earlier such as the communicators between management and query nodes.

Future Work

This paper has not discussed issues with updating to the OLAP structure. Many issues exist, specially in a situation like this when backups have to be computed. A strategy could rely on having a changes vector, where records are either deleted or appended, queries would be checked (by each query node) against the changes vector in order to add or remove records from the result set. When the changes vector reaches a stage of inefficiency, a new re computation of the backups may be of order. Such a scheme needs to be analyzed in a running OLAP system in order to determine the size capacity of such a changes vector, and to determine when would be a good time to recompute backups. Also periods of inactivity at the query nodes can be considered as good opportunities to recompute backups, as to not disrupt the service.

Conclusion

This paper has proposed a design for an OLAP machine that may handle k-failures. This has been shown through the means of coded redundancy, using Reed Solomon Codes, along with an MPI guideline on fault tolerant implementation issues. This paper discussed the proposed approach to handle a single view in OLAP; However, doing so can be replicated for all views, using the same approach. In conclusion, using the concepts outlined in this paper, an efficient fault tolerant approach can be applied to OLAP.

References

[1] Treaster M. A survey of fault-tolerance and fault-recovery techniques in parallel systems. 2004; .

[4] Gropp W, Lusk E. Fault tolerance in MPI programs. 2002; .

[2] Hamilton C. Fault tolerant parallel OLAP. 2005; Dalhousie Parallel Computing 2005: .

[3] Lamport and Shostak and Pease. The Byzantine Generals Problem. In: Anonymous Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), {IEEE} Computer Society Press. 1995: .

[4] Gropp W, Lusk E. Fault tolerance in MPI programs. 2002; .

[5] Reed IS, Solomon G. Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics 1960; 8: 300-304.

[6] Plank J. GFLIB - C procedures and programs for galois-field arithmetic and reed-solomon coding. 2003; 1.2: .

[7] Plank JS. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. 1996; .

Appendix

Pseudo Code for OLAP Algorithm in MPI

```
main(int argc, char * argv [ ] )
{
      int i, my_rank, p, k;
      MPI_Comm *my_comm;

      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
      MPI_Comm_size( MPI_COMM_WORLD, &p );
      k = p;

      /* create intercommunicators and set error handlers */
      if(my_rank == 0) //manager process
      {
          my_comm = (MPI_Comm*)malloc(p*sizeof(MPI_Comm));
          for ( i = 1; i < currsize; i++ )
          {
             MPI_Intercomm_create( MPI_COMM_SELF, 0, MPI_COMM_WORLD, i,
                                   IC_CREATE_TAG, &my_comm[i-1]);
             MPI_Comm_set_errhandler( my_comm[i-1], MPI_ERRORS_RETURN );
          }
      }
      else //other processes
      {
          my_comm = (MPI_Comm*)malloc(sizeof(MPI_Comm));
          MPI_Intercomm_create(MPI_COMM_SELF, 0, MPI_COMM_WORLD, 0, IC_CREATE_TAG,
                               my_comm);
          MPI_Comm_set_errhandler(*my_comm, MPI_ERRORS_RETURN );
      }

      /*    now each p has a communicator with the manager process     */

      if(not manager)
      {
          Data = Get_My_OLAP_Record_Partition();
          //this can be elaborated into send/recv pairs to allow the manager to
distribute
          Backup_Array = RS_encode_data(&Data, n = p-1-m , m);
          MPI_send_to_manager();
          MPI_recv_other_backups();
          //Write_backups_to_local_file();
      }
      else
      {
          Gather_Backup_data();
          Distribute_Backup_data();
      }

      if(manager)
      {
          get_olap_query();
          //in the send query phase, dead nodes will be detected.
          send_query_to_nodes(dead_node_marker [ ] );
          if(all_alive)
```

```
        {
           send_go_signal();
           gather_answer();
        }
        else
        {
           for(dead_nodes)
                 recover_data[i] = gather_dead_nodes_data();
           //recover_data now contains the subset of the original records
           //distribute records on processes + regather backups (to be modulated)
        }
     }
     else
     {
        receive_query();
        receive_message();
        if(message==go) answer_query();
        else
        {
           parse_message();
           get_RS_backup_of_dead_nodes();
           for(dead_nodes) send_backups_to_manager();
           //receive new record set + append to my set + recompute backups
        }
        send_answer_to_manager();
     }

     /*                      program complete             */

     if(my_rank==0) for ( i = 1; i < currsize; i++ ) MPI_Comm_free( &my_comm[i-
1] );
     else MPI_Comm_free(my_comm);

     MPI_Finalize( );
     return 0;
}
```