Volume 5, No. 2, March 2014 (Special Issue)
ISSN No. 0976-5697
International Journal of Advanced Research in Computer Science
RESEARCH PAPER
Available Online at www.ijarcs.info

Asynchronous Data Access and Transaction Decomposition in Distributed Databases

Dr N.Srinivasu
Associate professor
K L University
srinivasu28@kluniversity.in

S.M.Gouse Samdani, L.Sai Kiran
K L University
gousemailbox@gmail.com
Saikiran02@ymail.com

Abstract: A RDBMS is usually accessed using blocking drivers like JDBC/ODBC which require clients to block and wait for the result of each query they issue. An asynchronous database access mechanism would eliminate the need for such blocking and greatly improve client performance. *Thread-Per-Connection* and *Thread Pooling* are two methods currently being used to provide this asynchrony. These methods require the use of multiple threads in the database server, which provides the clients with access to the database. This is inefficient since a lot of memory and computing power is spent in creating, scheduling and switching multiple threads. In this paper, we show how asynchronous database access can be achieved with a single thread using the *Fork-Join mechanism* which employs *Future* objects in Java. We also show how *Asynchronous Transaction Decomposition in distributed databases* can help improve client performance.

Keywords: Asynchronous, Transaction Decomposition, Future objects, Fork-Join

I. INTRODUCTION

Relational Database Management Systems (RDBMS) are the predominant means for storing and retrieving data nowadays due to their relative ease of use compared to traditional hierarchical and network database systems. A relational database stores and operates on data using relations which are implemented as tables. Using SQL commands, a client can create, delete, retrieve and update data in these tables.

Database drivers like *ODBC* [1] and *JDBC* [2] serve as an interface between a RDBMS and its clients. These drivers are *blocking* in nature i.e. they require clients to block and wait for the result of each query they issue before being able to issue the next query. This affects the clients' performance as a lot of useful CPU time is wasted in such waits. If the queries are dispatched asynchronously, the clients can return immediately without blocking and then retrieve the results at a later time. One way of doing this is to create a new database connection for each query the client issues. This method requires the database server, which handles these connections, to follow a *Thread-Per-Connection* architecture in which a new thread is created and maintained for each such connection. Since server memory is limited, threads cannot be created infinitely as the number of queries issued and/or the number of clients increase and as a result, this architecture *does not scale well*.

Thread Pooling (shown in Fig 1) is a scalable variant of Thread-Per-Connection in which a fixed pool of worker threads service requests from a client. *The queries issued by a client are queued* and the client thread returns immediately without blocking. The worker threads then fetch the requests from this queue and dispatch them to the database. The maximum number of outstanding requests in the queue is bounded by the amount of memory available and can be greater than the number of executing threads. Hence, even if all worker threads are busy, additional requests can be held in the queue until any of the worker threads becomes available again.

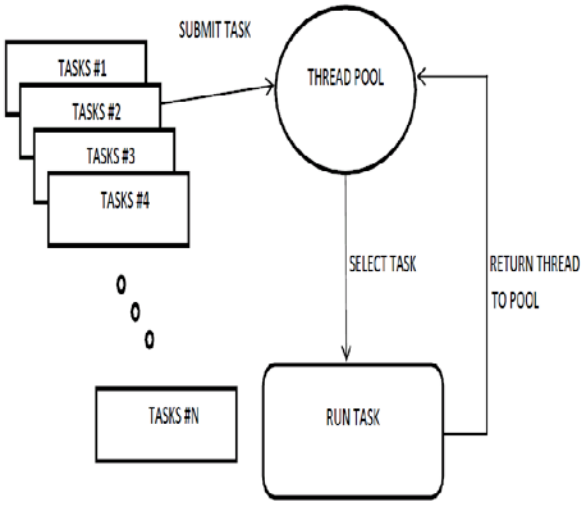

Figure 1. Thread Pooling

Both Thread-Per-Connection and Thread Pooling require the use of a *large number of threads per client* to achieve asynchronous database access. This is inefficient since creating and maintaining several threads is costly both in terms of the amount of server memory used up and in terms of the processor time spent in switching and scheduling the threads. Hence, there is a need to develop efficient methods for issuing queries asynchronously using a *minimal number of threads.*

In Section 2 of this paper we examine some proposed methods for asynchronous data access. Section 3 gives a detailed look at our solution (which uses a single thread per client) and explains why using a thread pool to service requests is inefficient. It also shows how the concept of transaction decomposition in distributed databases works. In Section 4, we analyze results from tests carried out on our system along with comparisons of the performance of our system versus other proposed methods under the same workload.

CONFERENCE PAPER
Two day National Conference on Advanced Trends and Challenges
in Computer Science and Applications
Organized by: Shree Vishnu Engineering College for Women, Bhimavaram A.P.
Schedule: 18-19 March 2014

II. RELATED WORK

A. *The Event Driven Model:*

One of the proposed alternatives to thread pooling is the *Event Driven Model*. Here, a *single thread* is used to handle multiple database connections. A Client mentions which database connection sockets it wishes to connect to and then issues its requests. Operating system buffers are used to store and transmit large data requests over these sockets in small packets. After the requests are serviced and the results are available, the *client is notified through socket events*.

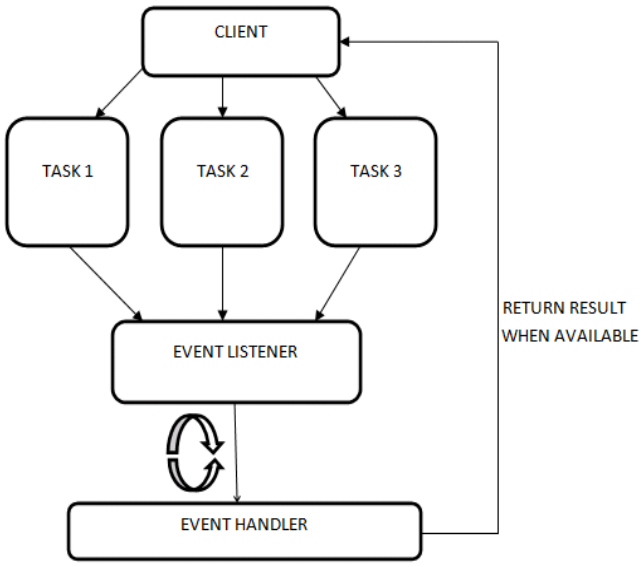

Figure 2. Event Driven Architecture

The event driven architecture scales well as it does not require multiple threads for asynchronous access. However, it is quite *complex to implement* and also *does not find support with existing database drivers*. Special drivers need to be developed to make this architecture work.

B. *Asynchronous Database Connectivity in Java (ADBCJ):*

Asynchronous Database Connectivity in Java (ADBCJ) [3] is a framework which uses *non-blocking socket I/O* along with a small number of OS threads to allow clients to issue queries to a RDBMS asynchronously. Clients use a *special API* for asynchronous RDBMS access to issue queries. ADBCJ notifies clients through *events* when the results are available. It also allows pipelining of RDBMS requests which further improves performance. One of the drawbacks with the current version of ADBCJ is that it *supports only simple data types* like integers, floating point numbers and strings. It also *works only with open source databases* like MySQL and PostgreSQL on Linux platforms.

III. ASYNCHRONOUS DB

A. *Fork-Join Mechanism:*

Our solution for asynchronous data access using a *single thread*, which we have named *Async DB*, makes use of the *Fork-Join mechanism* [4]. This is a parallel programming method based on the *divide and conquer strategy* in which a single task forks multiple subtasks, waits for their completion and then joins the results returned by the subtasks to obtain the final solution. The fork join mechanism can be used effectively with a single thread or a pool of threads. In our system, we use a single thread to issue queries from the client asynchronously and then allow the clients to obtain the results at a later time.

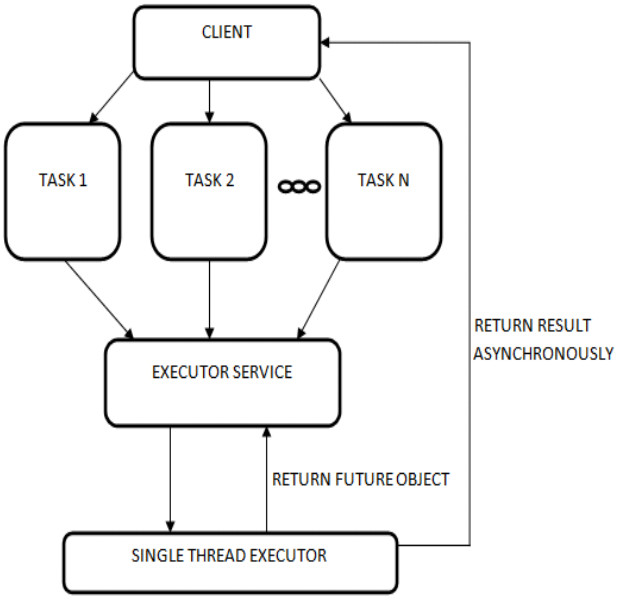

Figure 3. Fork-Join Mechanism using *Future* Objects

The Fork-Join mechanism is simpler to implement compared to the Event Driven model. One of the main advantages of Fork-Join over Event Driven programming is with respect to how *thread safety* is ensured. In Event Driven programming, the user is responsible for writing thread safe code whereas in Fork-Join, thread safety is handled automatically by the *Java Virtual Machine*. Failure to ensure thread safety is the reason why *Connection reset errors* are often noted to occur when ADBCJ (which makes use of event driven programming) is used with PostgreSQL databases under heavy loads.

B. *Interface Future in Java:*

The interface *Future* [5] is available under the *java.util.concurrent* package in Java 5.0. It is used along with the *ExecutorService* interface available in the same package for asynchronous and parallel computation in Java.

```
ExecutorService es = Executors.newSingleThreadExecutor();
Future<ResultSet> Result1 = es.submit(new QueryExecutor(Arguments));
Future<ResultSet> Result2 = es.submit(new QueryExecutor(Arguments));
Future<ResultSet> Result3 = es.submit(new QueryExecutor(Arguments));
..
..
public class QueryExecutor implements Callable<ResultSet>
{
        public QueryExecutor(Arguments) throws Exception
        {
                //Initialize Variables
        }
        public ResultSet call() throws Exception
        {
                //Perform Required Operations
                return ResultSet;
        }
}
```

Figure 4. Java code snippet showing how queries are issued asynchronously

Fig 4 shows a sample code snippet which uses the Future interface. An object of type Future represents the result of an asynchronous operation. The Executor Service interface [6] , [7] provided in Java 5 allows us to work with threads in a much easier way compared to the thread manipulation options provided in earlier versions of Java. The *newSingleThreadExecutor()* method returns an Executor which uses a single worker thread working off an unbounded queue. It ensures that tasks are executed sequentially and that no more than one task is active at a given time. The ExecutorService takes a query, which the client wishes to issue, in the form of a *Callable* object and returns a *Future* object. The single worker thread executes the query and when a result is available, it notifies the ExecutorService which then updates the Future object. The client has a variety of methods it can call on the Future object to perform operations like checking for completion of a query, cancelling the execution of a query, retrieving the result of a computation etc.

C. *Single Thread vs*:

Thread Pool It is possible to use an Executor which works with a fixed pool of threads to provide asynchronous data access capability but as mentioned in the introduction of this paper, we have found that *using a single thread is more efficient*. Fig 5 shows the comparison between an executor using fixed thread pools of three different sizes: 10, 100 and 1000 threads per pool respectively and an executor using a single thread with respect to the response time for executing a certain number of queries concurrently.

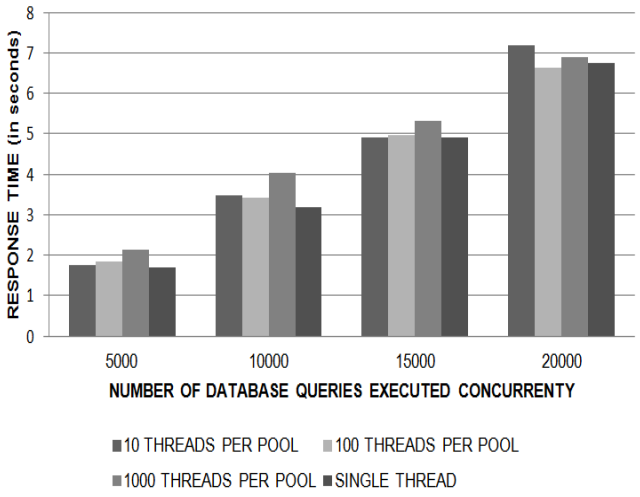

Figure 5. Performance comparison between a single thread and thread pools of various sizes

On an average, *the single thread executor is found to be 6% faster than the thread pool executor* when a high speed network with delay less than 1 ms is used. Using a single thread also provides great benefit in terms of the memory and processing power saved and helps service a greater number of clients compared to a thread pool. The following example clearly shows this.

Assume that the default stack space occupied by a JVM 1.6 thread in the server is 256 KB. The maximum memory allotted for the JVM's functioning in the server is 512 MB and of this, about 128 MB is used on an average for the JVM's internal structures, profiler agent code etc and 256 MB is allotted as heap memory for object creation at run time. That leaves 128 MB for non heap storage like thread stack space, loaded classes and other Meta data. If the total non heap memory is considered as the maximum stack space, then the maximum number of threads that can be created is 512. Using a thread pool which comprises three threads to service a batch of requests from a user, a maximum of 170 users can access the database concurrently. However, a single thread executor allocates just one thread per user and hence, up to 512 users are allowed concurrent access to the database.

D. *System Architecture:*

Async DB uses a *distributed database* system. Distributed databases have a number of useful characteristics like reliability, the ability to process huge workloads and fast data access from which present day database intensive consumer services like online ticket booking, online banking etc can greatly benefit from. We have taken advantage of these benefits to introduce *Asynchronous Transaction Decomposition* which we believe can improve end user performance many fold.

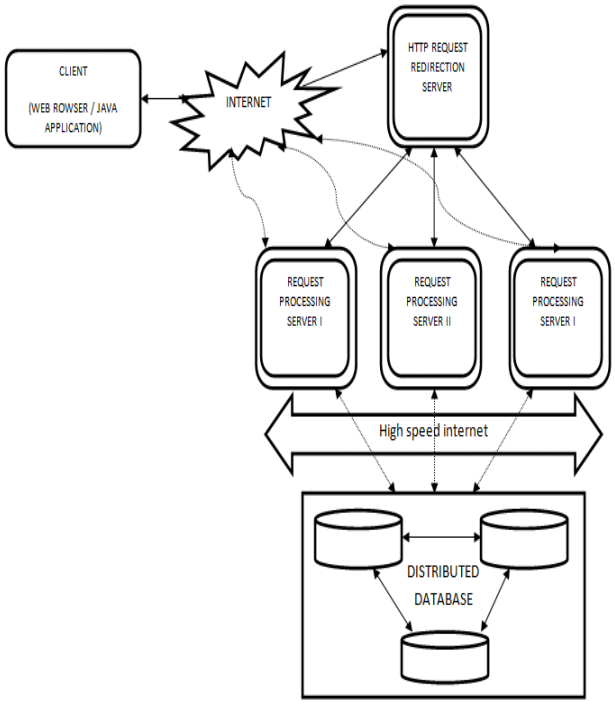

Figure 6. Asynchronous DB System Architecture

The architecture of our system is shown in Fig 6. The client is any web browser/desktop application which provides an interface for querying the database system. There can be any number of nodes connected together to form the distributed database system. In our implementation, we have restricted the size to three nodes, comprising the databases *Oracle*, *DB2* and *MySQL*. All the database copies contain the same data and a replication scheme is used to keep the data on these copies consistent. Unlike ADBCJ, our asynchronous data access method can work with *both commercial and open source databases* which are evident from the heterogeneous nature of Async DB [8]. There is also support for advanced data types like *binary objects*. *IBM's Web sphere Application Server Community Edition (WASCE)* serves as the database server and there is one server per each copy of the database. *Query execution is location based* [9] i.e. when a client queries or updates any table in the database, the request is sent to that server which

is found to be the nearest in terms of geographical location to the client. This ensures quick data access. Requests are redirected to whichever alternate nodes are available in the event the original node the user tries to access is busy or has crashed.

E. Asynchronous Transaction Decomposition (ADT):

Transaction decomposition is one of the features suggested by researchers to improve the transaction performance [10] , [11], [12], [13]. In Async DB, we make this method more efficient by introducing asynchrony in query execution. The queries in a transaction can be split into groups based on the dependencies existing between the queries such that no query in a group is dependent upon a query in any other group. We can then make use of the parallelism provided by a distributed database and the capability of asynchronous execution provided by Future objects to execute each group on a different database copy asynchronously. The following example clearly shows how transaction decomposition works. Consider these three tables used for booking a flight ticket online [14].

Customer (CustomerID, CustomerName, FreeMiles)
Flight (FlightNo, Origin, Destination, JourneyDistance, AvlTickets)
CustFlight (FlightNo, CustomerID, CustomerName)

The process of booking a ticket comprises of the following three activities in a transaction. Each activity comprises a number of sub-activities (For the sake of simplicity, we have left out the banking transactions in this example)

a. Blocking a Ticket:

Obtain a particular flight number from the customer.
(a). Read *AvlTickets* from *Flight* for this flight number.
(b). If *AvlTickets* > 0 , *AvlTickets* = *AvlTickets* - 1.

b. Updating the Airline Database with customer details:

(a). Read *CustomerName* and *CustomerID* from *Customer*.
(b). Update *CustFlight* with customer's details.

c. Updating the Frequent Flier Miles for the customer:

(a). Read *JourneyDistance* from *Flight*.
(b). Update *Customer* with a fraction of *JourneyDistance* as *FreeMiles*.

Here, activities 2 and 3 both depend on the success of activity 1 but are independent of each other. Hence after 1 completes successfully, 2 and 3 can be executed in parallel. This will result in the transaction competing much faster than if the activities were carried out sequentially. ADT can be made fail-safe by ensuring that the transaction commits only after the success of all three operations. The transaction is rolled back if any of the three operations fails.

IV. RESULTS

A. Asynchronous vs. Synchronous Data Access:

In order to test the efficiency of Async DB over traditional synchronous access, we executed a batch of retrieval queries using both techniques and observed the response times. This test was repeated for a total of five trials using a network with average communication delay of 1 ms .The observed response times are listed in Table 1.

Table 1. Observed response times for asynchronous and synchronous data access

Mode	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Asynch	3.94377	3.40204	3.57142	2.84677	2.85517
Synch	34.3694	39.7153	31.9314	34.5868	39.2095

From the results, we see that *our asynchronous data access method is on average about twelve times faster than traditional synchronous data access*. This strongly supports our view that using an asynchronous mechanism to access a database is much more efficient than using blocking connections. Fig 7 is a graphical representation of the data listed in Table 1 which clearly shows that clients can gain immense performance benefits by adopting the Async DB approach of accessing databases.

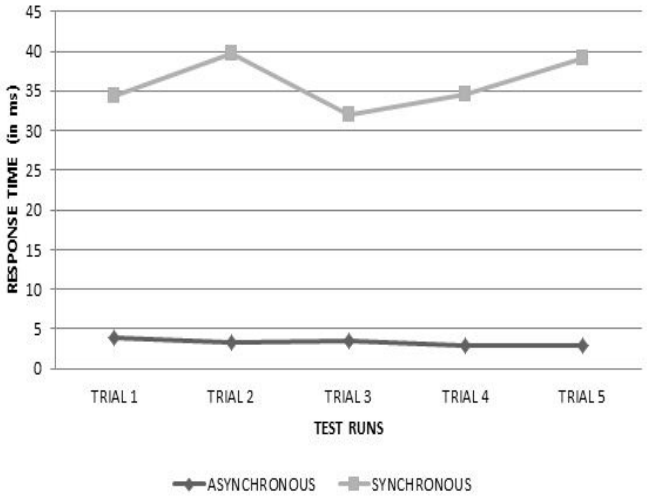

Figure 7. Performance comparison between asynchronous and synchronous data access

V. CONCLUSION

In this paper, we have shown how asynchronous execution of queries can provide immense performance benefits to database clients. Queries are currently executed by traditional database drivers using synchronous techniques which require clients to block and thereby waste a lot of CPU time until results are made available. Asynchronous data access removes the need to block clients, thereby allowing them to work on other operations. Some techniques like thread pooling, which have been adopted to provide asynchrony, require the use of multiple threads which is inefficient. Our proposed asynchronous data access strategy Async DB uses just a single thread and has been found to outperform alternatives like Event Driven Architecture and ADBCJ in terms of performance. Finally, we discuss how Asynchronous Transaction Decomposition (ADT) can greatly speed up transactions, which otherwise take up a lot of time to execute due to sequential execution of their component queries.

CONFERENCE PAPER
Two day National Conference on Advanced Trends and Challenges in Computer Science and Applications
Organized by: Shree Vishnu Engineering College for Women, Bhimavaram A.P.
Schedule: 18-19 March 2014

VI. REFERENCES

[1]. Microsoft: ODBC–Open Database Connectivity Overview. Web http://support. microsoft.com /kb/110093 (2007) Accessed December 2011

[2]. Oracle: JDBC 4.0 API Specification Final Release. Web http://java.sun.com/products/jdbc/ download.html (2006) Accessed December 2011

[3]. Heath, M.: Asynchronous Database Drivers. MS Thesis, Brigham Young University Web. http://contentdm.lib.byu.edu/ETD/image/etd4130.pdf (2011) Accessed December 2011

[4]. Goetz, B.: Java theory and practice: Stick a fork in it, Part 1. Web http://www.ibm.com/developerworks /java/library/j-jtp11137/index.html (2007) Accessed January 2012

[5]. Oracle: Interface Future. Web http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent /Future.html (2009) Accessed January 2012

[6]. Amis Technology Blog: Asynchronous processing in Java applications – leveraging those multi-cores. Web http://technology.amis.nl/2009/02/19/asynchronous-processing-in-java-applications-leveraging-those-multi-cores (2009) Accessed February 2012

[7]. Bloch, J.: Task Execution. In: Bloch, J., Bowbeer, J., Goetz, B., Holmes, D., Lea, D., Peierls, T. Java Concurrency in Practice, pp. 113-134. Addison Wesley Professional (2006)

[8]. Hepner, P.: Integrating Heterogeneous Databases: An Overview. Web ftp://deakin.edu.au/pub /TR/Computing/TRC9530.ps.gz (1999) Accessed February 2012

[9]. Cardellini, V., Colajanni, M., Yu, P.S.: Request Redirection Algorithms for Distributed Web Systems. IEEE Transactions on Parallel and Distributed Systems **14**(4), 355-368 (2003)

[10]. Mackinnon, L.M., Marwick, D.H., Williams, M.H.: A Model for Query Decomposition and Answer Construction in Heterogeneous Distributed Database Systems. Journal of Intelligent Information Systems **11**(1), 69-87 (1998)

[11]. Moss, E.: Nested Transactions: An Approach to Reliable Distributed Computing. MIT/ LCS/ TR-260, MIT Press (1985).

[12]. Haveman J.: Transaction Decomposition: Refinement of Timing Constraints. In: Proceedings of the South Pacific Conference on Formal Methods (1997)

[13]. Bernstein, A.J., Lewis, P.M: Transaction Decomposition Using Transaction Semantics. Distributed and Parallel Databases (1996)

[14]. Transaction Example - A Simple SQL Query. Web http://softbase.uwaterloo.ca/~tozsu/courses/cs448/notes/8.Transactions-ho.pdf Accessed February 2012